
PsychoPy - Psychology software for Python

Release 2022.2.3

Open Science Tools Ltd

Aug 12, 2022

CONTENTS

1	About	1
2	General issues	3
3	Installation	31
4	Getting Started	35
5	Builder	43
6	Coder	105
7	Running and sharing studies online	125
8	Communicating with external hardware using PsychoPy	147
9	Reference Manual (API)	149
10	Timing information for PsychoPy	779
11	Troubleshooting	781
12	Alerts	785
13	Recipes (“How-to”s)	803
14	Frequently Asked Questions (FAQs)	817
15	Resources (e.g. for teaching)	821
16	For Developers	823
17	Experiment file format (.psyexp)	851
	Python Module Index	855
	Index	857

1.1 Citing

If you use this software, please cite one of the publications that describe it. For most people **the 2019 paper is probably the most relevant** (the papers from 2009, 2007 did not mention Builder at all, for instance).

- Peirce, J. W., Gray, J. R., Simpson, S., MacAskill, M. R., Höchenberger, R., Sogo, H., Kastman, E., Lindeløv, J. (2019). *PsychoPy2: experiments in behavior made easy*. *Behavior Research Methods*. 10.3758/s13428-018-01193-y
- Peirce, J. W., Hirst, R. J. & MacAskill, M. R. (2022). *Building Experiments in PsychoPy*. 2nd Edn London: Sage.
- Peirce J. W. (2009). Generating stimuli for neuroscience using PsychoPy. *Frontiers in Neuroinformatics*, **2** (10), 1-8. doi:10.3389/neuro.11.010.2008
- Peirce, J. W. (2007). PsychoPy - Psychophysics software in Python. *Journal of Neuroscience Methods*, **162** (1-2):8-13 doi:10.1016/j.jneumeth.2006.11.017

Citing these papers gives the reviewer/reader of your study information about how the system works and it attributes some credit for its original creation. Academic assessment (whether for promotion or even getting appointed to a job in the first place) prioritises publications over making useful tools for others. Citations provide a way for the developers to justify their continued involvement in the development of the package.

1.2 License for use

is licensed under a [GPL3 license](#) which means, essentially, that:

- you can use it (and adapt it) for free in your work, and you can even release those versions
- but you must include the original license
- AND you must also make your release open source using the same license

What that means is you're free to use PsychoPy's goodwill in being open source, you are required to pass on that goodwill!

GENERAL ISSUES

These are issues that users should be aware of, whether they are using Builder or Coder views.

2.1 Monitor Center



provides a simple and intuitive way for you to calibrate your monitor and provide other information about it and then import that information into your experiment.

Information is inserted in the Monitor Center (Tools menu), which allows you to store information about multiple monitors and keep track of multiple calibrations for the same monitor.

For experiments written in the Builder view, you can then import this information by simply specifying the name of the monitor that you wish to use in the *Experiment settings* dialog. For experiments created as scripts you can retrieve the information when creating the *Window* by simply naming the monitor that you created in Monitor Center. e.g.:

```
from psychopy import visual
win = visual.Window([1024,768], mon='SonyG500')
```

Of course, the name of the monitor in the script needs to match perfectly the name given in the Monitor Center.

2.1.1 Real world units

One of the particular features of is that you can specify the size and location of stimuli in units that are independent of your particular setup, such as degrees of visual angle (see *Units for the window and stimuli*). In order for this to be possible you need to inform of some characteristics of your monitor. Your choice of units determines the information you need to provide:

Units	Requires
'norm' (normalised to width/height)	n/a
'pix' (pixels)	Screen width in pixels
'cm' (centimeters on the screen)	Screen width in pixels and screen width in cm
'deg' (degrees of visual angle)	Screen width (pixels), screen width (cm) and distance (cm)

2.1.2 Calibrating your monitor

can also store and use information about the gamma correction required for your monitor. If you have a Spectrascan PR650, PR655/PR670, Minolta LS100/LS110 or a CRS ColorCAL you can perform an automated calibration in which will measure the necessary gamma value to be applied to your monitor. Alternatively this can be added manually into the grid to the right of the Monitor Center. To run a calibration, connect the photometer via the serial port and, immediately after turning it on press the *Get Photometer* button in the Monitor Center.

Note that, if you don't have a photometer to hand then there is a method for determining the necessary gamma value psychophysically included in (see `gammaMotionNull` and `gammaMotionAnalysis` in the coder demos menu, under "experiment control").

The two additional tables in the Calibration box of the Monitor Center provide conversion from *DKL* and *LMS* colour spaces to *RGB*.

2.2 Units for the window and stimuli

One of the key advantages of over many other experiment-building software packages is that stimuli can be described in a wide variety of real-world, device-independent units. In most other systems you provide the stimuli at a fixed size and location in pixels, or percentage of the screen, and then have to calculate how many cm or degrees of visual angle that was.

In , after providing information about your monitor, via the *Monitor Center*, you can simply specify your stimulus in the unit of your choice and allow to calculate the appropriate pixel size for you.

Your choice of unit depends on the circumstances. For conducting demos, the two normalised units ('norm' and 'height') are often handy because the stimulus scales naturally with the window size. For running an experiment it's usually best to use something like 'cm' or 'deg' so that the stimulus is a fixed size irrespective of the monitor/window.

For all units, the centre of the screen is represented by coordinates (0,0), negative values mean down/left, positive values mean up/right.

For help understanding spatial units visually, try the builder demo "`spatialUnits`" under "Understanding PsychoPy" (version 2021.2).

2.2.1 Units for online experiments

If you are running a study online the easiest units to use will be those that require no monitor info. It is likely that your experiment will be run on a wide variety of devices all with differing screen resolutions. Furthermore it is going to be more difficult for you to control factors like viewing distance. Because of this it makes it difficult to use units like *deg* or *cm* - because we need to know both the participants viewing distance and the number of pixels that make up a cm on that participants screen. The easiest solution here is to use *Height units*, this means that the size of stimuli will be scaled relative to the height of that participants screen - which usually means it is possible to run studies even on smartphones!

Note: If using height units on a tablet/touchscreen device, currently 1 unit height corresponds to the height of the screen when the device is held in landscape.

Degrees of visual angle are not currently supported for online use, but you can estimate pixels per cm using a [screen scaling method](#) (this demo was shared by Wakefield Morys Carter 2021) and then use pixel units to present stimuli in cm see [Li et al \(2020\)](#) for more details. If you want to store the window size of your participants device in an online study, you can add a code component and use `thisExp.addData('windowSize', win.size)`.

2.2.2 Height units

With ‘height’ units everything is specified relative to the height of the window (note the window, not the screen). As a result, the dimensions of a screen with standard 4:3 aspect ratio will range (-0.6667,-0.5) in the bottom left to (+0.6667,+0.5) in the top right. For a standard widescreen (16:10 aspect ratio) the bottom left of the screen is (-0.8,-0.5) and top-right is (+0.8,+0.5). This type of unit can be useful in that it scales with window size, unlike *Degrees of visual angle* or *Centimeters on screen*, but stimuli remain square, unlike *Normalised units* units. Obviously it has the disadvantage that the location of the right and left edges of the screen have to be determined from a knowledge of the screen dimensions. (These can be determined at any point by the *Window.size* attribute.)

Spatial frequency: cycles **per stimulus** (so will scale with the size of the stimulus).

Requires :No monitor information

2.2.3 Normalised units

In normalised (‘norm’) units the window ranges in both x and y from -1 to +1. That is, the top right of the window has coordinates (1,1), the bottom left is (-1,-1). Note that, in this scheme, setting the height of the stimulus to be 1.0, will make it half the height of the window, not the full height (because the window has a total height of 1:-1 = 2!). Also note that specifying the width and height to be equal will not result in a square stimulus if your window is not square - the image will have the same aspect ratio as your window. e.g. on a 1024x768 window the size=(0.75,1) will be square.

Spatial frequency: cycles **per stimulus** (so will scale with the size of the stimulus).

Requires : No monitor information

2.2.4 Centimeters on screen

Set the size and location of the stimulus in centimeters on the screen.

Spatial frequency: cycles per cm

Requires : information about the screen width in cm and size in pixels

Assumes : pixels are square. Can be verified by drawing a stimulus with matching width and height and verifying that it is in fact square. For a *CRT* this can be controlled by setting the size of the viewable screen (settings on the monitor itself).

2.2.5 Degrees of visual angle

Use degrees of visual angle to set the size and location of the stimulus. This is, of course, dependent on the distance that the participant sits from the screen as well as the screen itself, so make sure that this is controlled, and remember to change the setting in *Monitor Center* if the viewing distance changes.

Spatial frequency: cycles per degree

Requires : information about the screen width in cm and pixels and the viewing distance in cm

There are actually three variants: ‘deg’, ‘degFlat’, and ‘degFlatPos’

- **‘deg’** : Most people using degrees of visual angle choose to make the assumption that a degree of visual angle spans the same number of pixels at all parts of the screen. This isn’t actually true for standard flat screens - a degree of visual angle at the edge of the screen spans more pixels because it is further from the eye. For moderate eccentricities the error is small (a 0.2% error in size calculation at 3 deg eccentricity) but grows as stimuli are placed further from the centre of the screen (a 2% error at 10 deg). For most studies this form of

calculation is preferred, as it does not result in a warped appearance of visual stimuli, but if you need greater precision at far eccentricities then choose one of the alternatives below.

- **‘degFlatPos’** : This accounts for flat screens in calculating position coordinates of visual stimuli but leaves size and spatial frequency uncorrected. This means that an evenly spaced grid of visual stimuli will appear warped in position but will
- **‘degFlat’**: This corrects the calculations of degrees for flatness of the screen for each vertex of your stimuli. Square stimuli in the periphery will, therefore, become more spaced apart but they will also get larger and rhomboid in the pixels that they occupy.

2.2.6 Pixels on screen

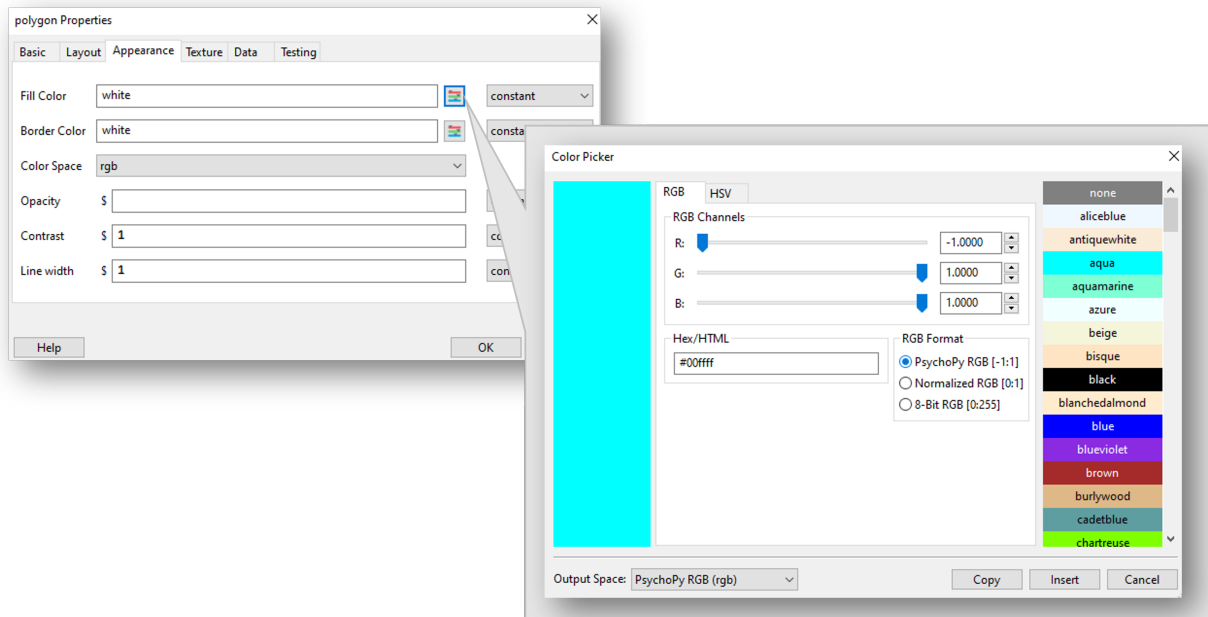
You can also specify the size and location of your stimulus in pixels. Obviously this has the disadvantage that sizes are specific to your monitor (because all monitors differ in pixel size).

Spatial frequency: ``cycles per pixel`` (this catches people out but is used to be in keeping with the other units. If using pixels as your units you probably want a spatial frequency in the range 0.2-0.001 (i.e. from 1 cycle every 5 pixels to one every 100 pixels).

Requires : information about the size of the screen (not window) in pixels, although this can often be deduce from the operating system if it has been set correctly there.

Assumes: nothing

2.3 Color spaces



You can explore colors in PsychoPy Builder through accessing the color picker from any parameter that takes a color value.

The color of stimuli can be specified when creating a stimulus and when using `setColor()` in a variety of ways. From Builder view you can also use the color picker to pick the color you want and explore what value that color would

correspond to in a variety of spaces. There are three basic color spaces that can use, RGB, DKL and LMS but colors can also be specified by a name (e.g. 'DarkSalmon') or by a hexadecimal string (e.g. '#00FF00').

examples:

```
stim = visual.GratingStim(win, color=[1,-1,-1], colorSpace='rgb') #will be red
stim.setColor('Firebrick')#one of the web/X11 color names
stim.setColor('#FFFAF0')#an off-white
stim.setColor([0,90,1], colorSpace='dkl')#modulate along S-cone axis in isoluminant_
↳plane
stim.setColor([1,0,0], colorSpace='lms')#modulate only on the L cone
stim.setColor([1,1,1], colorSpace='rgb')#all guns to max
stim.setColor([1,0,0])#this is ambiguous - you need to specify a color space
```

2.3.1 Colors by name

Any of the [web/X11 color names](#) can be used to specify a color. These are then converted into RGB space by .

These are not case sensitive, but should not include any spaces.

2.3.2 Colors by hex value

This is really just another way of specifying the r,g,b values of a color, where each gun's value is given by two hexadecimal characters. For some examples see [this chart](#). To use these in they should be formatted as a string, beginning with # and with no spaces. (NB on a British Mac keyboard the # key is hidden - you need to press Alt-3)

2.3.3 RGB color space

This is the simplest color space, in which colors are represented by a triplet of values that specify the red green and blue intensities. These three values each range between -1 and 1.

Examples:

- [1, 1, 1] is white
- [0, 0, 0] is grey
- [-1, -1, -1] is black
- [1.0, -1, -1] is red
- [1.0, 0.6, 0.6] is pink

The reason that these colors are expressed ranging between 1 and -1 (rather than 0:1 or 0:255) is that many experiments, particularly in visual science where has its roots, express colors as deviations from a grey screen. Under that scheme a value of -1 is the maximum decrement from grey and +1 is the maximum increment above grey.

You can still specify colors in RGB from 0:1 or 0:255, but you will need to let know that this is what you're doing. To do this, set the color space to be *rgb1* for 0:1 or *rgb255* for 0:255 - if the color space is just *rgb*, then values will be from -1:1

Note that will use your monitor calibration to linearize this for each gun. E.g., 0 will be halfway between the minimum luminance and maximum luminance for each gun, if your monitor gammaGrid is set correctly.

2.3.4 HSV color space

Another way to specify colors is in terms of their Hue, Saturation and ‘Value’ (HSV). For a description of the color space see the [Wikipedia HSV entry](#). The Hue in this case is specified in degrees, the saturation ranging 0:1 and the ‘value’ also ranging 0:1.

Examples:

- `[0, 1, 1]` is red
- `[0, 0.5, 1]` is pink
- `[90, 1, 1]` is cyan
- `[anything, 0, 1]` is white
- `[anything, 0, 0.5]` is grey
- `[anything, anything, 0]` is black

Note that colors specified in this space (like in RGB space) are not going to be the same another monitor; they are device-specific. They simply specify the intensity of the 3 primaries of your monitor, but these differ between monitors. As with the RGB space gamma correction is automatically applied if available.

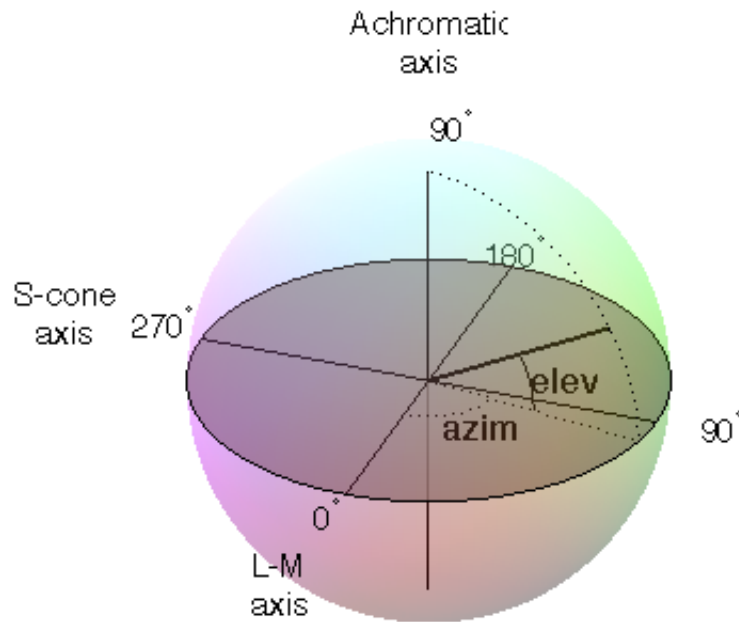
2.3.5 DKL color space

To use DKL color space the monitor should be calibrated with an appropriate spectrophotometer, such as a PR650.

In the Derrington, Krauskopf and Lennie¹ color space (based on the Macleod and Boynton² chromaticity diagram) colors are represented in a 3-dimensional space using spherical coordinates that specify the *elevation* from the isoluminant plane, the *azimuth* (the hue) and the contrast (as a fraction of the maximal modulations along the cardinal axes of the space).

¹ Derrington, A.M., Krauskopf, J., & Lennie, P. (1984). Chromatic Mechanisms in Lateral Geniculate Nucleus of Macaque. *Journal of Physiology*, 357, 241-265.

² MacLeod, D. I. A. & Boynton, R. M. (1979). Chromaticity diagram showing cone excitation by stimuli of equal luminance. *Journal of the Optical Society of America*, 69(8), 1183-1186.



In these values are specified in units of degrees for elevation and azimuth and as a float (ranging -1:1) for the contrast. Note that not all colors that can be specified in DKL color space can be reproduced on a monitor. You can see [here a movie plotting colors in DKL space](#) (showing *cartesian* coordinates, not spherical coordinates) to show the gamut of colors available on an example monitor.

Examples:

- $[90, 0, 1]$ is white (maximum elevation aligns the color with the luminance axis)
- $[0, 0, 1]$ is an isoluminant stimulus, with azimuth 0 (S-axis)
- $[0, 45, 1]$ is an isoluminant stimulus, with an oblique azimuth

2.3.6 LMS color space

To use LMS color space the monitor should be calibrated with an appropriate spectrophotometer, such as a PR650.

In this color space you can specify the relative strength of stimulation desired for each cone independently, each with a value from -1:1. This is particularly useful for experiments that need to generate cone isolating stimuli (for which modulation is only affecting a single cone type).

2.4 Preferences

The Preferences dialog allows to adjust general settings for different parts of . The preferences settings are saved in the configuration file *userPrefs.cfg*. The labels in brackets for the different options below represent the abbreviations used in the *userPrefs.cfg* file.

In rare cases, you might want to adjust the preferences on a per-experiment basis. See the API reference for the *Preferences class here*.

2.4.1 General settings (General)

window type (winType): can use one of two ‘backends’ for creating windows and drawing; pygame, pyglet and glfw. Here you can set the default backend to be used.

units (units): Default units for windows and visual stimuli (‘deg’, ‘norm’, ‘cm’, ‘pix’). See *Units for the window and stimuli*. Can be overridden by individual experiments.

full-screen (fullscr): Should windows be created full screen by default? Can be overridden by individual experiments.

allow GUI (allowGUI): When the window is created, should the frame of the window and the mouse pointer be visible. If set to False then both will be hidden.

paths (paths): Paths for additional Python packages can be specified. See more *information on paths here*.

flac audio compression (flac): Set flac audio compression.

parallel ports (parallelPorts): This list determines the addresses available in the drop-down menu for the *Parallel Port Out Component*.

2.4.2 Application settings (App)

These settings are common to all components of the application (Coder and Builder etc)

show start-up tips (showStartupTips): Display tips when starting .

large icons (largeIcons): Do you want large icons (on some versions of wx on macOS this has no effect)?

default view (defaultView): Determines which view(s) open when the app starts up. Default is ‘last’, which fetches the same views as were open when last closed.

reset preferences (resetPrefs): Reset preferences to defaults on next restart of .

auto-save prefs (autoSavePrefs): Save any unsaved preferences before closing the window.

debug mode (debugMode): Enable features for debugging itself, including unit-tests.

locale (locale): Language to use in menus etc.; not all translations are available. Select a value, then restart the app. Think about *adding translations for your language*.

2.4.3 Builder settings (Builder)

reload previous exp (reloadPrevExp): Select whether to automatically reload a previously opened experiment at start-up.

uncluttered namespace (unclutteredNamespace): If this option is selected, the scripts will use more complex code, but the advantage is that there is less of a chance that name conflicts will arise.

components folders (componentsFolders): A list of folder path names that can hold additional custom components for the Builder view; expects a comma-separated list.

hidden components (hiddenComponents): A list of components to hide (e.g., because you never use them)

unpacked demos dir (unpackedDemosDir): Location of Builder demos on this computer (after unpacking).

saved data folder (savedDataFolder): Name of the folder where subject data should be saved (relative to the script location).

Flow at top (topFlow): If selected, the “Flow” section will be shown topmost and the “Components” section will be on the left. Restart to activate this option.

always show readme (alwaysShowReadme): If selected, always shows the Readme file if you open an experiment. The Readme file needs to be located in the same folder as the experiment file.

max favorites (maxFavorites): Upper limit on how many components can be in the Favorites menu of the Components panel.

2.4.4 Coder settings (Coder)

code font (codeFont): A list of font names to be used for code display. The first found on the system will be used.

comment font (commentFont): A list of font names to be used for comments sections. The first found on the system will be used

output font (outputFont): A list of font names to be used in the output panel. The first found on the system will be used.

code font size (codeFontSize): An integer between 6 and 24 that specifies the font size for code display in points.

output font size (outputFontSize): An integer between 6 and 24 that specifies the font size for output display in points.

show source asst (showSourceAsst): Do you want to show the source assistant panel (to the right of the Coder view)? On Windows this provides help about the current function if it can be found. On macOS the source assistant is of limited use and is disabled by default.

show output (showOutput): Show the output panel in the Coder view. If shown all python output from the session will be output to this panel. Otherwise it will be directed to the original location (typically the terminal window that called application to open).

reload previous files (reloadPrevFiles): Should fetch the files that you previously had open when it launches?

preferred shell (preferredShell): Specify which shell should be used for the coder shell window.

newline convention (newlineConvention): Specify which character sequence should be used to encode newlines in code files: unix = n (line feed only), dos = rn (carriage return plus line feed).

2.4.5 Connection settings (Connections)

proxy (proxy): The proxy server used to connect to the internet if needed. Must be of the form `http://111.222.333.444:5555`

auto-proxy (autoProxy): should try to deduce the proxy automatically. If this is True and autoProxy is successful, then the above field should contain a valid proxy address.

allow usage stats (allowUsageStats): Allow to ping a website at when the application starts up. Please leave this set to True. The info sent is simply a string that gives the date, version and platform info. There is no cost to you: no data is sent that could identify you and will not be delayed in starting as a result. The aim is simple: if we can show that lots of people are using there is a greater chance of it being improved faster in the future.

check for updates (checkForUpdates): can (hopefully) automatically fetch and install updates. This will only work for minor updates and is still in a very experimental state (as of v1.51.00).

timeout (timeout): Maximum time in seconds to wait for a connection response.

2.4.6 Hardware settings

audioLib : Select your choice of audio library with a list of names specifying the order they should be tried. We recommend [`'PTB'`, `'sounddevice'`, `'pyo'`, `'pygame'`] for lowest latency.

audioLatencyMode [0, 1, 2, 3, 4] Latency mode for PsychToolbox audio (3 is good for most applications. See *PTB Audio Latency Modes*).

audioDriver: 'portaudio' Some of PsychoPy's audio engines provide the option not to use portaudio but go directly to another lib (e.g. to coreaudio) but some don't allow that

```
# audio driver to use audioDriver = list(default=list('portaudio')) # audio device to use (if audi-
oLib allows control) audioDevice = list(default=list('default')) # a list of parallel ports parallelPorts =
list(default=list('0x0378', '0x03BC')) # The name of the Qmix pump configuration to use qmixConfiguration =
string(default='qmix_config')
```

2.4.7 Key bindings

There are many shortcut keys that you can use in . For instance did you realise that you can indent or outdent a block of code with Ctrl-[and Ctrl-] ?

2.5 Data outputs

There are a number of different forms of output that can generate, depending on the study and your preferred analysis software. Multiple file types can be output from a single experiment (e.g. *Excel data file* for a quick browse, *Log file* to check for error messages and *data file (.psydat)* for detailed analysis)

2.5.1 Log file

Log files are actually rather difficult to use for data analysis but provide a chronological record of everything that happened during your study. The level of content in them depends on you. See *Logging data* for further information.

2.5.2 data file (.psydat)

This is actually a *TrialHandler* or *StairHandler* object that has been saved to disk with the python *cPickle* module.

.psydat files can be useful for retrieving data that you forgot to explicitly tell to save. They can also be more directly used by experienced users with previous experience of python and, probably, matplotlib. The contents of the file can be explored with `dir()`, as any other python object.

.psydat files are ideal for batch analysis with a python script and plotting via *matplotlib*. They contain more information than the Excel or csv data files, and can even be used to (re)create those files.

Of particular interest might be the following attributes and methods of the Handler:

entries a list of dictionaries. Each entry/dictionary in the list represents a single trial's (a single routine 'run'/iteration) data.

saveAsPickle() a method for saving all of the entries' data in a Python pickle file

saveAsWideText() a method for saving all of the entrie's data in a .csv file.

If you just want to recover data or first wish to try things out in a familiar format you can put all of the data in a .csv file, very similar to the .csv files that are produced by default when running experiments. The following script assumes you're using a command-line interface (e. g. Terminal on Mac, or the Command Prompt on Windows) where you've opened up a Python shell, and that you have installed as a Python package:

```
# import PsychoPy function for loading Pickle/JSON data
from psychopy.misc import fromFile
# (replace with the file path to your .psydat file)
fpath = '/Users/my_user/myexperiments/myexperiment/participant_expname_date.psydat'
# load in the data
psydata = fromFile(fpath)
# (replace with the file path to where you want the resulting .csv
# to be saved)
save_path = '/Users/my_user/Desktop/test_out.csv'
# save the data as a .csv file, ie separating the values with a
# comma. 'CSV' simply means 'comma-separated values'
psydata.saveAsWideText(save_path, delim=',')
```

To get started using the data directly in Python, you can try opening a psydat file and printing all its entries:

```
from psychopy.misc import fromFile
# (replace with the file path to your .psydat file)
fpath = 'path/to/file.psydat'
psydata = fromFile(fpath)
for entry in psydata.entries:
    print(entry)
```

Ideally, we should provide a demo script here for fetching and plotting some data (feel free to *contribute*).

2.5.3 Long-wide data file

This form of data file is the default data output from Builder experiments as of v1.74.00. Rather than summarising data in a spreadsheet where one row represents all the data from a single condition (as in the summarised data format), in long-wide data files the data is not collapsed by condition, but written chronologically with one row representing one trial (hence it is typically longer than summarised data files). One column in this format is used for every single piece of information available in the experiment, even where that information might be considered redundant (hence the format is also ‘wide’).

Although these data files might not be quite as easy to read quickly by the experimenter, they are ideal for import and analysis under packages such as R, SPSS or Matlab.

2.5.4 Excel data file

Excel 2007 files (.xlsx) are a useful and flexible way to output data as a spreadsheet. The file format is open and supported by nearly all spreadsheet applications (including older versions of Excel and also OpenOffice). N.B. because .xlsx files are widely supported, the older Excel file format (.xls) is not likely to be supported by unless a user contributes the code to the project.

Data from are output as a table, with a header row. Each row represents one condition (trial type) as given to the *TrialHandler*. Each column represents a different type of data as given in the header. For some data, where there are multiple columns for a single entry in the header. This indicates multiple trials. For example, with a standard data file in which response time has been collected as ‘rt’ there will be a heading *rt_raw* with several columns, one for each trial that occurred for the various trial types, and also an *rt_mean* heading with just a single column giving the mean reaction time for each condition.

If you’re creating experiments by writing scripts then you can specify the sheet name as well as file name for Excel file outputs. This way you can store multiple sessions for a single subject (use the subject as the filename and a date-stamp as the sheetname) or a single file for multiple subjects (give the experiment name as the filename and the participant as the sheetname).

Builder experiments use the participant name as the file name and then create a sheet in the Excel file for each loop of the experiment. e.g. you could have a set of practice trials in a loop, followed by a set of main trials, and these would each receive their own sheet in the data file.

2.5.5 Delimited text files (.csv, .tsv, .txt)

For maximum compatibility, especially for legacy analysis software, you can choose to output your data as a delimited text file. Typically this would be comma-separated values (.csv file) or tab-delimited (.tsv file). The format of those files is exactly the same as the Excel file, but is limited by the file format to a single sheet.

2.6 Gamma correcting a monitor

Monitors typically don’t have linear outputs; when you request luminance level of 127, it is not exactly half the luminance of value 254. For experiments that require the luminance values to be linear, a correction needs to be put in place for this nonlinearity which typically involves fitting a power law or gamma (γ) function to the monitor output values. This process is often referred to as gamma correction.

can help you perform gamma correction on your monitor, especially if you have one of the supported photometers/spectroradiometers.

There are various different equations with which to perform gamma correction. The simple equation (2.1) is assumed by most hardware manufacturers and gives a reasonable first approximation to a linear correction. The full gamma

correction equation (2.3) is more general, and likely more accurate especially where the lowest luminance value of the monitor is bright, but also requires more information. It can only be used in labs that do have access to a photometer or similar device.

2.6.1 Simple gamma correction

The simple form of correction (as used by most hardware and software) is this:

$$L(V) = a + kV^\gamma \quad (2.1)$$

where L is the final luminance value, V is the requested intensity (ranging 0 to 1), a , k and γ are constants for the monitor.

This equation assumes that the luminance where the monitor is set to ‘black’ ($V=0$) comes entirely from the surround and is therefore not subject to the same nonlinearity as the monitor. If the monitor itself contributes significantly to a then the function may not fit very well and the correction will be poor.

The advantage of this function is that the calibrating system (in this case) does not need to know anything more about the monitor than the gamma value itself (for each gun). For the full gamma equation (2.3), the system needs to know about several additional variables. The look-up table (LUT) values required to give a (roughly) linear luminance output can be generated by:

$$LUT(V) = V^{1/\gamma} \quad (2.2)$$

where V is the entry in the LUT, between 0 (black) and 1 (white).

2.6.2 Full gamma correction

For very accurate gamma correction uses a more general form of the equation above, which can separate the contribution of the monitor and the background to the lowest luminance level:

$$L(V) = a + (b + kV)^\gamma \quad (2.3)$$

This equation makes no assumption about the origin of the base luminance value, but requires that the system knows the values of b and k as well as γ .

The inverse values, required to build the LUT are found by:

$$LUT(V) = \frac{((1 - V)b^\gamma + V(b + k)^\gamma)^{1/\gamma} - b}{k} \quad (2.4)$$

This is derived below, for the interested reader. ;-)

And the associated luminance values for each point in the LUT are given by:

$$L(V) = a + (1 - V)b^\gamma + V(b + k)^\gamma$$

2.6.3 Deriving the inverse full equation

The difficulty with the full gamma equation (2.3) is that the presence of the b value complicates the issue of calculating the inverse values for the LUT. The simple inverse of (2.3) as a function of output luminance values is:

$$LUT(L) = \frac{((L - a)^{1/\gamma} - b)}{k} \quad (2.5)$$

To use this equation we need to first calculate the linear set of luminance values, L , that we are able to produce the current monitor and lighting conditions and *then* deduce the LUT value needed to generate that luminance value.

We need to insert into the LUT the values between 0 and 1 (to use the maximum range) that map onto the linear range from the minimum, m , to the maximum M possible luminance. From the parameters in (2.3) it is clear that:

$$\begin{aligned} m &= a + b^\gamma \\ M &= a + (b + k)^\gamma \end{aligned} \tag{2.6}$$

Thus, the luminance value, L at any given point in the LUT, V , is given by

$$\begin{aligned} L(V) &= m + (M - m)V \\ &= a + b^\gamma + (a + (b + k)^\gamma - a - b^\gamma)V \\ &= a + b^\gamma + ((b + k)^\gamma - b^\gamma)V \\ &= a + (1 - V)b^\gamma + V(b + k)^\gamma \end{aligned} \tag{2.7}$$

where V is the position in the LUT as a fraction.

Now, to generate the LUT as needed we simply take the inverse of (2.3):

$$LUT(L) = \frac{(L - a)^{1/\gamma} - b}{k} \tag{2.8}$$

and substitute our $L(V)$ values from (2.7):

$$\begin{aligned} LUT(V) &= \frac{(a + (1 - V)b^\gamma + V(b + k)^\gamma - a)^{1/\gamma} - b}{k} \\ &= \frac{((1 - V)b^\gamma + V(b + k)^\gamma)^{1/\gamma} - b}{k} \end{aligned} \tag{2.9}$$

2.6.4 References

2.7 OpenGL and Rendering

All rendering performed by uses hardware-accelerated [OpenGL](#) rendering where possible. This means that, as much as possible, the necessary processing to calculate pixel values is performed by the graphics card *GPU* rather than by the *CPU*. For example, when an image is rotated the calculations to determine what pixel values should result, and any interpolation that is needed, are determined by the graphics card automatically.

In the double-buffered system, stimuli are initially drawn into a piece of memory on the graphics card called the ‘back buffer’, while the screen presents the ‘front buffer’. The back buffer initially starts blank (all pixels are set to the window’s defined color) and as stimuli are ‘rendered’ they are gradually added to this back buffer. The way in which stimuli are combined according to transparency rules is determined by the *blend mode* of the window. At some point in time, when we have rendered to this buffer all the objects that we wish to be presented, the buffers are ‘flipped’ such that the stimuli we have been drawing are presented simultaneously. The monitor updates at a very precise fixed rate and the flipping of the window will be synchronised to this monitor update if possible (see [Sync to VBL and wait for VBL](#)).

Each update of the window is referred to as a ‘frame’ and this ultimately determines the temporal resolution with which stimuli can be presented (you cannot present your stimulus for any duration other than a multiple of the frame duration). In addition to synchronising flips to the frame refresh rate, can optionally go a further step of not allowing the code to continue until a screen flip has occurred on the screen, which is useful in ascertaining exactly when the frame refresh occurred (and, thus, when your stimulus actually appeared to the subject). These timestamps are very precise on most computers. For further information about synchronising and waiting for the refresh see [Sync to VBL and wait for VBL](#).

If the code/processing required to render all your stimuli to the screen takes longer to complete than one screen refresh then you will ‘drop/skip a frame’. In this case the previous frame will be left on screen for a further frame period and the flip will only take effect on the following screen update. As a result, time-consuming operations such as disk accesses or execution of many lines of code, should be avoided while stimuli are being dynamically updated (if you care about the precise timing of your stimuli). For further information see the sections on *Detecting dropped frames* and *Reducing dropped frames*.

2.7.1 Fast and slow functions

The fact that modern graphics processors are extremely powerful; they can carry out a great deal of processing from a very small number of commands. Consider, for instance, the Coder demo *elementArrayStim* in which several hundred Gabor patches are updated frame by frame. The graphics card has to blend a sinusoidal grating with a grey background, using a Gaussian profile, several hundred times each at a different orientation and location and it does this in less than one screen refresh on a good graphics card.

There are three things that are relatively slow and should be avoided at critical points in time (e.g. when rendering a dynamic or brief stimulus). These are:

1. disk accesses
2. passing large amounts of data to the graphics card
3. making large numbers of python calls.

Functions that are very fast:

1. Calls that move, resize, rotate your stimuli are likely to carry almost no overhead
2. Calls that alter the color, contrast or opacity of your stimulus will also have no overhead IF your graphics card supports *OpenGL Shaders*
3. Updating of stimulus parameters for `psychopy.visual.ElementArrayStim` is also surprisingly fast BUT you should try to update your stimuli using *numpy* arrays for the maths rather than *for... loops*

Notable slow functions in PsychoPy calls:

1. Calls to set the image or set the mask of a stimulus. This involves having to transfer large amounts of data between the computer’s main processor and the graphics card, which is a relatively time-consuming process.
2. Any of your own code that uses a Python *for... loop* is likely to be slow if you have a large number of cycles through the loop. Try to ‘vectorise’ your code using a *numpy* array instead.

2.7.2 Tips to render stimuli faster

1. Keep images as small as possible. This is meant in terms of **number of pixels**, not in terms of Mb on your disk. Reducing the size of the image on your disk might have been achieved by image compression such as using jpeg images but these introduce artefacts and do nothing to reduce the problem of sending large amounts of data from the CPU to the graphics card. Keep in mind the size that the image will appear on your monitor and how many pixels it will occupy there. If you took your photo using a 10 megapixel camera that means the image is represented by 30 million numbers (a red, green and blue) but your computer monitor will have, at most, around 2 megapixels (1960x1080).
2. Try to use square powers of two for your image sizes. This is efficient because computer memory is organised according to powers of two (did you notice how often numbers like 128, 512, 1024 seem to come up when you buy your computer?). Also several mathematical routines (anything involving Fourier maths, which is used a lot in graphics processing) are faster with power-of-two sequences. For the *psychopy.visual.GratingStim* a texture/mask of this size is **required** and if you don’t provide one then your texture will be

‘upsampled’ to the next larger square-power-of-2, so you can save this interpolation step by providing it in the right shape initially.

3. Get a faster graphics card. Upgrading to a more recent card will cost around £30. If you’re currently using an integrated Intel graphics chip then almost any graphics card will be an advantage. Try to get an nVidia or an ATI Radeon card.

2.7.3 OpenGL Shaders

You may have heard mention of ‘shaders’ on the users mailing list and wondered what that meant (or maybe you didn’t wonder at all and just went for a donut!). OpenGL shader programs allow modern graphics cards to make changes to things during the rendering process (i.e. while the image is being drawn). To use this you need a graphics card that supports OpenGL 2.1 and will only make use of shaders if a specific OpenGL extension that allows floating point textures is also supported. Nowadays nearly all graphics cards support these features - even Intel chips from Intel!

One example of how such shaders are used is the way that colors greyscale images. If you provide a greyscale image as a 128x128 pixel texture and set its color to be red then, without shaders, needs to create a texture that contains the 3x128x128 values where each of the 3 planes is scaled according to the RGB values you require. If you change the color of the stimulus a new texture has to be generated with the new weightings for the 3 planes. However, with a shader program, that final step of scaling the texture value according to the appropriate RGB value can be done by the graphics card. That means we can upload just the 128x128 texture (taking 1/3 as much time to upload to the graphics card) and then we each time we change the color of the stimulus we just a new RGB triplet (only 3 numbers) without having to recalculate the texture. As a result, on graphics cards that support shaders, changing colors, contrasts and opacities etc. has almost zero overhead.

2.7.4 Blend Mode

A ‘blend function’ determines how the values of new pixels being drawn should be combined with existing pixels in the ‘frame buffer’.

blendMode = ‘avg’

This mode is exactly akin to the real-world scenario of objects with varying degrees of transparency being placed in front of each other; increasingly transparent objects allow increasing amounts of the underlying stimuli to show through. Opaque stimuli will simply occlude previously drawn objects. With each increasing semi-transparent object to be added, the visibility of the first object becomes increasingly weak. The order in which stimuli are rendered is very important since it determines the ordering of the layers. Mathematically, each pixel colour is constructed from $opacity * stimRGB + (1 - opacity) * backgroundRGB$. This was the only mode available before version 1.80 and remains the default for the sake of backwards compatibility.

blendMode = ‘add’

If the window *blendMode* is set to ‘add’ then the value of the new stimulus does not in any way *replace* that of the existing stimuli that have been drawn; it is added to it. In this case the value of *opacity* still affects the weighting of the new stimulus being drawn but the first stimulus to be drawn is never ‘occluded’ as such. The sum is performed using the signed values of the color representation in , with the mean grey being represented by zero. So a dark patch added to a dark background will get even darker. For grating stimuli this means that contrast is summed correctly.

This blend mode is ideal if you want to test, for example, the way that subjects perceive the sum of two potentially overlapping stimuli. It is also needed for rendering stereo/dichoptic stimuli to be viewed through colored anaglyph glasses.

If stimuli are combined in such a way that an impossible luminance value is requested of any of the monitor guns then that pixel will be out of bounds. In this case the pixel can either be clipped to provide the nearest possible colour, or can be artificially colored with noise, highlighting the problem if the user would prefer to know that this has happened.

2.7.5 Sync to VBL and wait for VBL

will always, if the graphics card allows it, synchronise the flipping of the window with the vertical blank interval (VBL aka VBI) of the screen. This prevents visual artefacts such as ‘tearing’ of moving stimuli. This does not, itself, indicate that the script also waits for the physical frame flip to occur before continuing. If the `waitBlanking` window argument is set to `False` then, although the window refreshes themselves will only occur in sync with the screen VBL, the `win.flip()` call will not actually wait for this to occur, such that preparations can continue immediately for the next frame. For rendering purposes this is actually optimal and will reduce the likelihood of frames being dropped during rendering.

By default the Window will also wait for the VBL (`waitBlanking=True`). Although this is slightly less efficient for rendering purposes it is necessary if we need to know exactly when a frame flip occurred (e.g. to timestamp when the stimulus was physically presented). On most systems this will provide a very accurate measure of when the stimulus was presented (with a variance typically well below 1ms but this should be tested on your system).

2.8 Timing Issues and synchronisation

One of the key requirements of experimental control software is that it has good temporal precision. aims to be as precise as possible in this domain and can achieve excellent results depending on your experiment and hardware. It also provides you with a precise log file of your experiment to allow you to check the precision with which things occurred.

Many scientists have asked “Can provide sub-millisecond timing precision?”. The short answer is yes it can - ‘s timing is as good as any software package we’ve tested ([we’ve tested quite a lot](#)).

BUT there are many components to getting good timing, and many ways that your timing could be less-than-perfect. So if timing is important to you then you should really read this entire section of the manual and you should **test your timing** using dedicated hardware (photodiodes, microphones or, ideally the [Black Box Toolkit](#)). We can’t emphasise enough how many ways there are for your hardware and/or operating system to break the good timing that is providing.

2.8.1 Can deliver millisecond precision?

The simple answer is ‘yes’. PsychoPy’s timing is as good as (or in most cases better than) any package out there. For detailed Studies of timing see [Bridges et al., 2020](#)

The longer answer is that you should test the timing of your own experimental stimuli on your own hardware. Very often a computer is not optimally configured to produce good timing, and a poorly coded experiment could also destroy your timing (which is one reason we now recommend even good coders use [Builder!](#)). Many software and hardware manufacturers will suggest that the key to good timing is using computer clocks with high precision (lots of decimal places) but that is not the answer at all. The sources of error in stimulus/response timing are almost never to do with the poor precision of the clock. The following issues are extremely common and **until you actually test your experiment you don’t realise that your timing is being affected by them:**

- *Additional delays caused by monitors:* e.g. the monitor taking additional time to process the image before presentation
- *Delays caused by drivers and OS:* Windows, Linux and Mac all perform further processing on the images, depending on settings and this can delay your visual stimulus delivery by a further frame interval or more
- *Delays caused by coding errors*

- *Delays caused by keyboards*
- *Audio delays*

The clocks that uses do have sub-millisecond precision but your keyboard has a latency of 4-25ms depending on your platform and keyboard. You could buy a response pad (e.g. a [Labhackers Millikey](#)) for response timing with a sub-ms precision, but note that there will still be an apparent lag that is dependent on the monitor's absolute lag and the position of the stimulus on it.

Also note that the variance in terms of response times between your participants, and from trial to trial within a participant, probably dwarfs that of your keyboard and monitor issues! That said, does aim to give you as high a temporal precision as possible and, in a well-configured system achieves this very well.

2.8.2 Computer monitors

There are several issues with monitors that you should be aware of.

1. *Monitors have fixed refresh rates*
2. *The top of the screen appears 5-15 ms before the bottom*
3. *Additional delays caused by monitors*

Monitors have fixed refresh rates

Most monitors have fixed refresh rates, typically 60 Hz for a flat-panel display. You probably knew that but it's very easy to forget that this means certain stimulus durations won't be possible. If your screen is a standard 60 Hz monitor then your frame period is 1/60 s, roughly 16.7 ms. That means you can generate stimuli that last for 16.7 ms, or 33.3 ms or 50 ms, but you **cannot** present a stimulus for 20, 40, or 60 ms.

The caveat to this is that you can now buy specialist monitors that support variable refresh rates (although not below at least 5 ms between refreshes). These are using a technology called G-Sync (nVidia) or FreeSync (everyone else) and can make use of those technologies but support isn't built in to the library. See the publication by [Poth et al \(2018\)](#) for example code.

The top of the screen appears 5-15 ms before the bottom

For most monitor technologies, the lines of pixels are drawn sequentially from the top to the bottom and once the bottom line has been drawn the screen is finished and the line returns to the top (the Vertical Blank Interval, VBI). Most of your frame interval is spent drawing the lines, with 1-2ms being left for the VBI. This means that the pixels at the bottom are drawn "up to 10 ms later" than the pixels at the top of the screen. At what point are you going to say your stimulus 'appeared' to the participant?

Additional delays caused by monitors

Monitors themselves often cause delays **on top of** the unavoidable issue of having a refresh rate. Modern displays often have features to optimize the image, which will be often labelled as modes like "Movie Mode", "Game Mode" etc. If your display has any such settings then you want to turn them off so as not to change your image. Not only do these settings entail altering the color of the pixels that your experiment generator is send to the screen (if you've spent time carefully calibrating your colors and then the monitor changes them it would be annoying) but these forms of "post-processing" take time and often a *variable* time.

If your monitor has any such "post-processing" enabled then you might well be seeing an additional 20-30 ms of (variable) lag added to the stimulus onset as a result. This **will not** be detected by psychoPy (or any other system) and will not show up in your log files.

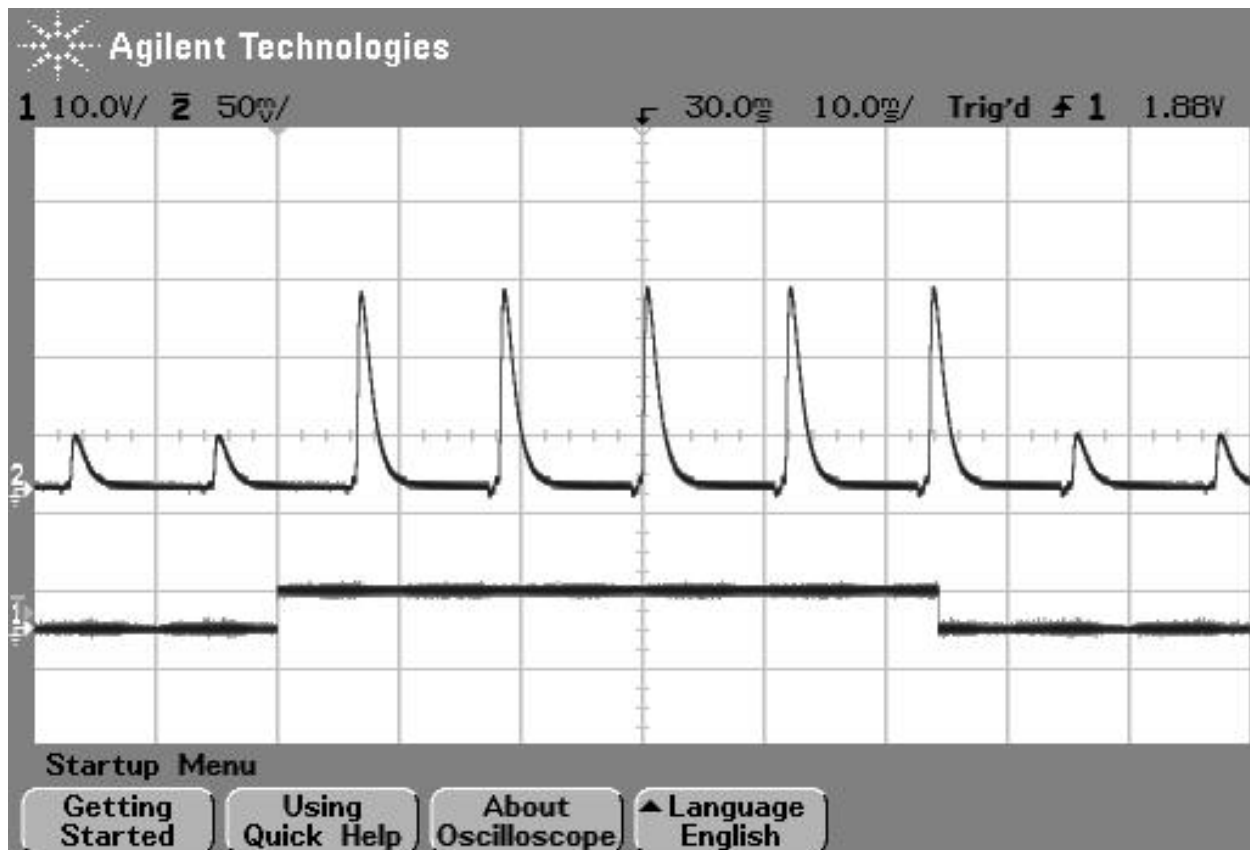


Fig. 2.1: Figure 1: photodiode trace at top of screen. The image above shows the luminance trace of a CRT recorded by a fast photo-sensitive diode at the top of the screen when a stimulus is requested (shown by the square wave). The square wave at the bottom is from a parallel port that indicates when the stimulus was flipped to the screen. Note that on a CRT the screen at any point is actually black for the majority of the time and just briefly bright. The visual system integrates over a large enough time window not to notice this. On the next frame after the stimulus ‘presentation time’ the luminance of the screen flash increased.

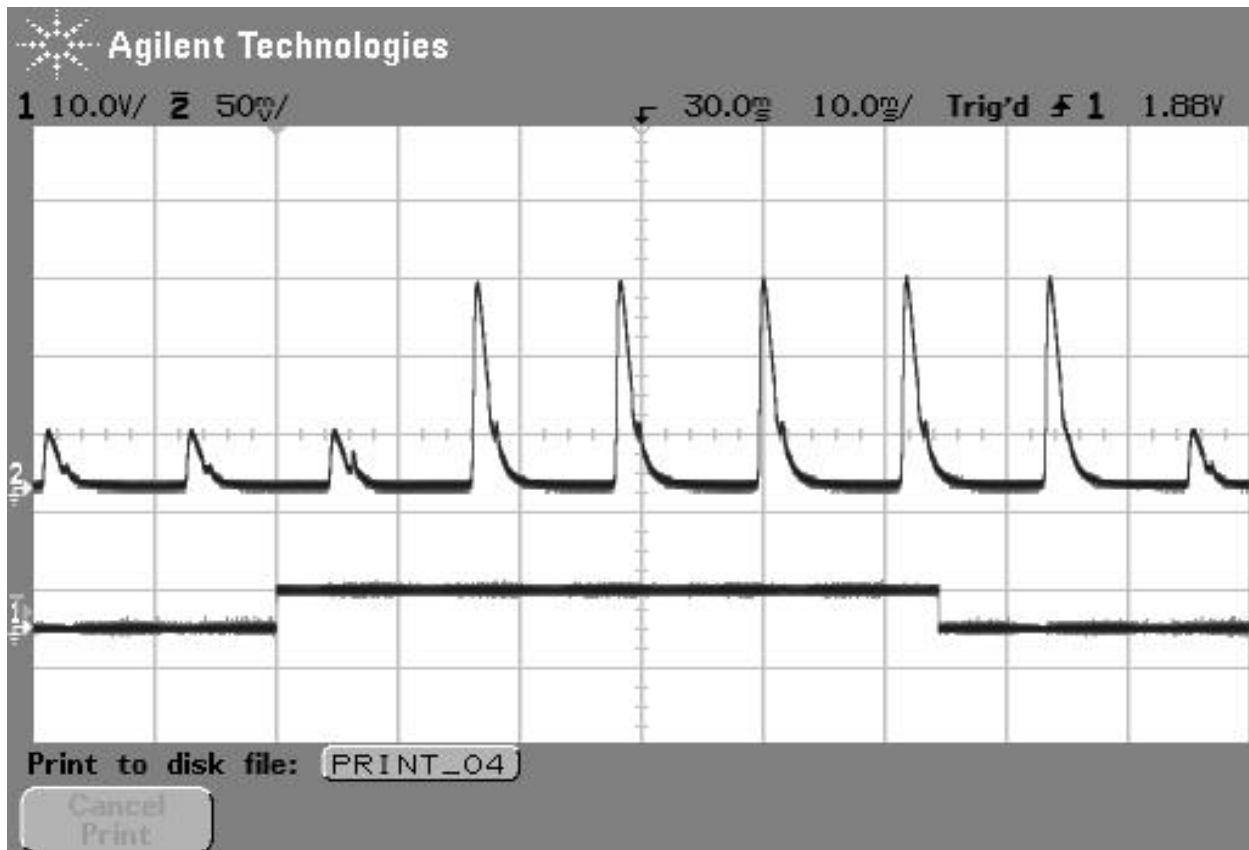


Fig. 2.2: Figure 2: photodiode trace of the same large stimulus at bottom of screen. The image above shows comes from exactly the same script as the above but the photodiode is positioned at the bottom of the screen. In this case, after the stimulus is 'requested' the current frame (which is dark) finishes drawing and then, 10ms later than the above image, the screen goes bright at the bottom.

2.8.3 Delays caused by drivers and OS

All three major operating systems are capable of introducing timing errors into your visual presentations, although these are usually observed as (relatively) constant lags. The particularly annoying factor here is that your experiment might work with very good timing for a while and then the operating system performs an automatic update and the timing gets worse! Again, the only way you would typically know about these sorts of changes is by testing with hardware.

Triple buffering: In general, and similar graphics systems, are expecting a double-buffered rendering pipeline, whereby we are drawing to one copy of the screen (the “back buffer”) and when we have finished drawing our stimuli we “flip” the screen, at which point it will wait for the next screen refresh period and become visible as the “front buffer”. Triple-buffering is a system whereby the images being rendered to the screen are put in a 3rd buffer, and the operating system can do further processing as the rendered image moves from this 3rd buffer to the back buffer. Such a system means that your images all appear exactly one frame later than expected.

Errors caused by triple buffering, either by the operating system or by the monitor, cannot be detected by and will not show up in your log files.

MacOS

The stimulus presentation on MacOS used to be very good, up until version 10.12. In MacOS 10.13 something changed and it appears that a form of triple buffering has been added and, to date, none of the major experiment generators have managed to turn this off. As a result, since MacOS 10.13 stimuli appear always to be presented a screen refresh period later than expected, resulting in a delay of 16.66 ms in the apparent response times to visual stimuli.

Windows 10

In Windows, triple buffering is something that might be turned on by default in your graphics card settings (look for 3D, or OpenGL, settings in the driver control panel to turn this off). The reason it gets used is that it often results in a more consistent frame rate for games, but having the frame appear later than expected is typically bad for experiments!

As well as the graphics card performing triple buffering, the operating system itself (via the Desktop Window Manager) does so under certain conditions: - Anytime a window is used (instead of full-screen mode) Windows 10 now uses triple buffering - having Scaling set to anything other than 100% also results in triple-buffering (presumably Microsoft renders the screen once and then scales it during the next refresh).

There are surely other settings in Windows and the graphics card that will alter the timing performance and, again, until you test these you aren't likely to know.

Linux

In Linux, again, timing performance of the visual stimuli depends on the graphics card driver but we have also seen timing issues arising from the Window Compositor and with interactions between compositor and driver.

The real complication here is that in Linux there are many different window compositors (Compiz, XFwm, Enlightenment, . . .), as well as different options for drivers to install (e.g. for nVidia cards there is a proprietary nVidia driver as well as an open-source “Nouveau” driver which is often the default but has worse performance).

Ultimately, you need to **test the timing with hardware** and work through the driver/compositor settings to optimise the timing performance.

2.8.4 Delays caused by coding errors

It can be really easy, as a user, to introduce timing errors into your experiment with incorrect coding. Even if you really know what you're doing, it's easy to make a silly mistake, and if you don't really know what you're doing then all bets are definitely off!!

Common ways for this to happen are to forget the operations that are potentially time-consuming. The biggest of these is the loading of images from disk.

For image stimuli where the image is constant the image should be loaded from disk at the beginning of the script (Builder-generate experiments will do so automatically for you). When an image stimulus has to *change on each trial*, it must be loaded from disk at some point. That typically takes several milliseconds (possibly hundreds of milliseconds for a large image) and while that is happening the screen will not be refreshing. You need to take your image-loading time into account and allow it to occur during a static period of the screen.

In *B Builder* experiments if you set something to update "On every repeat" then it will update as that Routine begins so, if your trial Routine simply begins with 0.5s fixation period, all your stimuli can be loaded/updated in that period and you will have no further problems. Sometimes you want to load/update your stimulus explicitly at a different point in time and then you can insert a *Static Component* Component into your Builder experiment (a "Static Period" in the Python API) and then set your stimulus to update during that period (it will show up as an update option after you insert the Static Component).

The good news is that a lot of the visual timing issues caused by coding problems **are** visible in the *Log Files*, unlike the problems with hardware and operating systems introducing lags.

2.8.5 Delays caused by keyboards

Keyboards are hopeless for timing. We should expand on that. But for now, it's all you need to know! Get yourself a button box, like the *LabHackers Millikey*.

2.8.6 Audio delays

has a number of settings for audio and the main issue here is that the user needs to know to turn on the optimal settings.

For years we were looking for a library that provided fast reliable audio and we went through an number of libraries to optimize that (pygame was the first, with 100ms latencies, then pyo and sounddevice which were faster).

Most recently we added support for the Psychophysics Toolbox audio library (PsychPortAudio), which Mario Kleiner has ported Python in 2018. With that library we can achieve really remarkable audio timing (thanks to Mario for his fantastic work). But still there are several things you need to check to make use of this library and use it to its full potential:

- Make sure you're running with a 64bit installation of Python3. The PsychPortAudio code has not, and almost certainly will not, be built to support legacy Python installations
- Set the preferences to use it! As of version 3.2.x the PTB backend was not the default. In future versions this will probably be the default, but as of version 3.2.x you need to set to use it (we didn't want to make it the default until it had been used without issue in a number of labs in "the wild").
- Make sure that the library settings are using a high

For further information please see the documentation about the *Sound library*

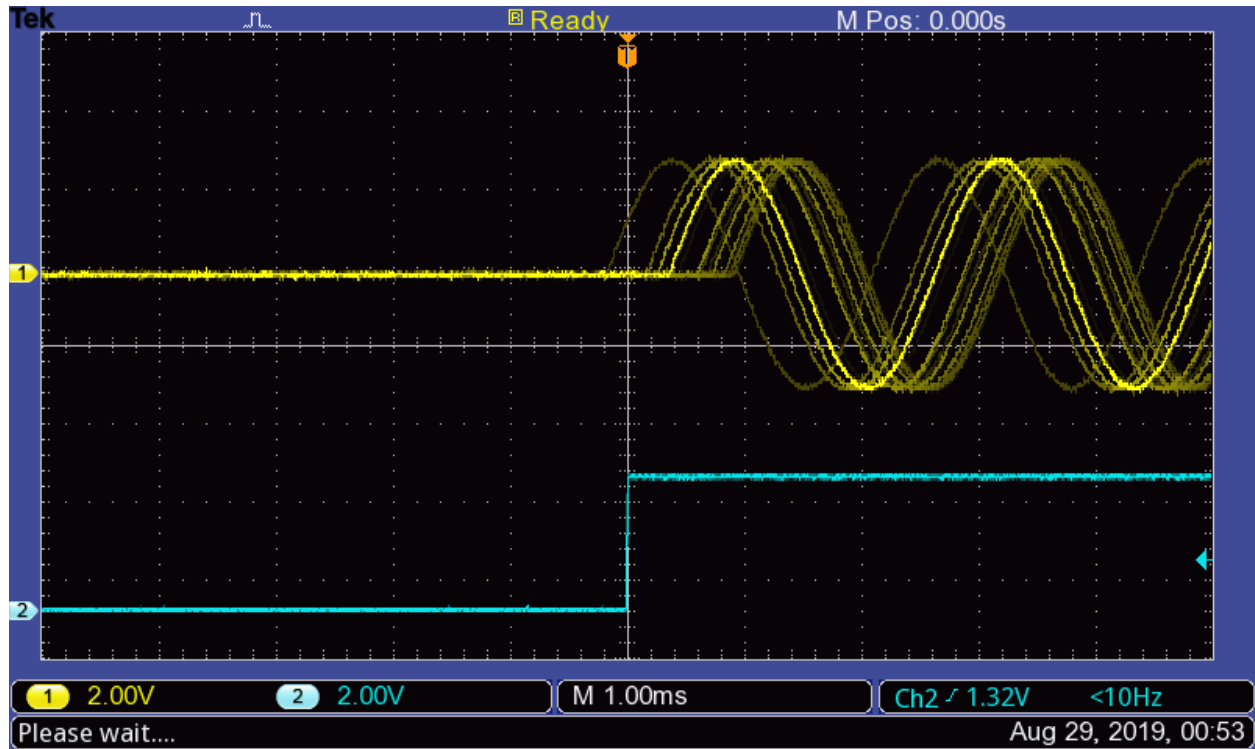


Fig. 2.3: With the new PTB library you can achieve not only sub-millisecond precision, but roughly sub-millisecond lags!! You do need to know how to configure this though and testing it can only be done with hardware.

2.8.7 Non-slip timing for imaging

For most behavioural/psychophysics studies timing is most simply controlled by setting some timer (e.g. a `Clock()`) to zero and waiting until it has reached a certain value before ending the trial. We might call this a ‘relative’ timing method, because everything is timed from the start of the trial/epoch. In reality this will cause an overshoot of some fraction of one screen refresh period (10ms, say). For imaging (EEG/MEG/fMRI) studies adding 10ms to each trial repeatedly for 10 minutes will become a problem, however. After 100 stimulus presentations your stimulus and scanner will be de-synchronised by 1 second.

There are two ways to get around this:

1. *Time by frames* If you are confident that you *aren't dropping frames* then you could base your timing on frames instead to avoid the problem.
2. *Non-slip (global) clock timing* The other way, which for imaging is probably the most sensible, is to arrange timing based on a global clock rather than on a relative timing method. At the start of each trial you add the (known) duration that the trial will last to a *global* timer and then wait until that timer reaches the necessary value. To facilitate this, the `Clock()` was given a new `add()` method as of version 1.74.00 and a `CountdownTimer()` was also added.

The non-slip method can only be used in cases where the trial is of a known duration at its start. It cannot, for example, be used if the trial ends when the subject makes a response, as would occur in most behavioural studies.

Non-slip timing from the Builder

When creating experiments in the *Builder*, will attempt to identify whether a particular *Routine* has a known end-point in seconds. If so then it will use non-slip timing for this Routine based on a global countdown timer called *routineTimer*. Routines that are able to use this non-slip method are shown in green in the *Flow*, whereas Routines using relative timing are shown in red. So, if you are using PsychoPy for imaging studies then make sure that all the Routines within your loop of epochs are showing as green. (Typically your study will also have a Routine at the start waiting for the first scanner pulse and this will use relative timing, which is appropriate).

2.8.8 Detecting dropped frames

Occasionally you will drop frames if you:

- try to do too much drawing
- do it in an inefficient manner (write poor code)
- have a poor computer/graphics card

Things to avoid:

- recreating textures for stimuli
- building new stimuli from scratch (create them once at the top of your script and then change them using `stim.setOri(ori)()`, `stim.setPos([x,y]..)`)

Turn on frame time recording

The key sometimes is *knowing* if you are dropping frames. can help with that by keeping track of frame durations. By default, frame time tracking is turned off because many people don't need it, but it can be turned on any time after *Window* creation:

```
from psychopy import visual
win = visual.Window([800,600])
win.recordFrameIntervals = True
```

Since there are often dropped frames just after the system is initialised, it makes sense to start off with a fixation period, or a ready message and don't start recording frame times until that has ended. Obviously if you aren't refreshing the window at some point (e.g. waiting for a key press with an unchanging screen) then you should turn off the recording of frame times or it will give spurious results.

Warn me if I drop a frame

The simplest way to check if a frame has been dropped is to get to report a warning if it thinks a frame was dropped:

```
from psychopy import visual, logging
win = visual.Window([800,600])

win.recordFrameIntervals = True

# By default, the threshold is set to 120% of the estimated refresh
# duration, but arbitrary values can be set.
#
# I've got 85Hz monitor and want to allow 4 ms tolerance; any refresh that
# takes longer than the specified period will be considered a "dropped"
```

(continues on next page)

(continued from previous page)

```
# frame and increase the count of win.nDroppedFrames.
win.refreshThreshold = 1/85 + 0.004

# Set the log module to report warnings to the standard output window
# (default is errors only).
logging.console.setLevel(logging.WARNING)

print('Overall, %i frames were dropped.' % win.nDroppedFrames)
```

Show me all the frame times that I recorded

While recording frame times, these are simply appended, every frame to `win.frameIntervals` (a list). You can simply plot these at the end of your script using `matplotlib`:

```
import matplotlib.pyplot as plt
plt.plot(win.frameIntervals)
plt.show()
```

Or you could save them to disk. A convenience function is provided for this:

```
win.saveFrameIntervals(fileName=None, clear=True)
```

The above will save the currently stored frame intervals (using the default filename, 'lastFrameIntervals.log') and then clears the data. The saved file is a simple text file.

At any time you can also retrieve the time of the /last/ frame flip using `win.lastFrameT` (the time is synchronised with `logging.defaultClock` so it will match any logging commands that your script uses).

'Blocking' on the VBI

As of version 1.62 'blocks' on the vertical blank interval meaning that, once `Window.flip()` has been called, no code will be executed until that flip actually takes place. The timestamp for the above frame interval measurements is taken immediately after the flip occurs. Run the `timeByFrames` demo in Coder to see the precision of these measurements on your system. They should be within 1ms of your mean frame interval.

Note that Intel integrated graphics chips (e.g. GMA 945) under `win32` do not sync to the screen at all and so blocking on those machines is not possible.

2.8.9 Reducing dropped frames

There are many things that can affect the speed at which drawing is achieved on your computer. These include, but are probably not limited to; your graphics card, CPU, operating system, running programs, stimuli, and your code itself. Of these, the CPU and the OS appear to make rather little difference. To determine whether you are actually dropping frames see [Detecting dropped frames](#).

Things to change on your system:

1. make sure you have a good graphics card. Avoid integrated graphics chips, especially Intel integrated chips and especially on laptops (because on these you don't get to change your mind so easily later). In particular, try to make sure that your card supports OpenGL 2.0
2. **shut down as many programs, including background processes. Although modern processors are fast and often have multiple cores, they are still limited by the amount of RAM and the amount of disk space available.**
 - anti-virus auto-updating (if you're allowed)
 - email checking software
 - file indexing software
 - backup solutions (e.g. TimeMachine)
 - Dropbox
 - Synchronisation software

Writing optimal scripts

1. run in full-screen mode (rather than simply filling the screen with your window). This way the OS doesn't have to spend time working out what application is currently getting keyboard/mouse events.
2. don't generate your stimuli when you need them. Generate them in advance and then just modify them later with the methods like `setContrast()`, `setOrientation()` etc. . .
3. **calls to the following functions are comparatively slow; they require more CPU time than most other functions and then have a large overhead.**
 - `GratingStim.setTexture()`
 - `RadialStim.setTexture()`
 - `TextStim.setText()`
4. if you don't have OpenGL 2.0 then calls to `setContrast`, `setRGB` and `setOpacity` will also be slow, because they also make a call to `setTexture()`. If you have shader support then this call is not necessary and a large speed increase will result.
5. avoid loops in your python code (use numpy arrays to do maths with lots of elements) *Note: numpy arrays will not work for online experiments, which use JavaScript**
6. if you need to create a large number (e.g. greater than 10) similar stimuli, then try the `ElementArrayStim` (currently not supported for online experiments)

Possible good ideas

It isn't clear that these actually make a difference, but they might).

1. disconnect the internet cable (to prevent programs performing auto-updates?)
2. on Macs you can actually shut down the Finder. It might help. See [Alex Holcombe's timing tips page](#)
3. use a single screen rather than two (probably there is some graphics card overhead in managing double the number of pixels?)

2.8.10 Understand and measuring your timing

There are certain steps that we strongly advise you to take before running an experiment that needs to be temporally precise in PsychoPy, or indeed any other software:

- Read [this timing megastudy](#) by Bridges et al (2020) which compares several pieces of behavioural software in terms of their temporal precision.
- Check that your stimulus presentation monitor is not dropping frames. You can do this by running the `timeByFrames.py` demo. Find this demo in the *Coder* window > demos > timing. The `timeByFrames` demo examines the precision of your frame flips, and shows the results in a plot similar to the one below:

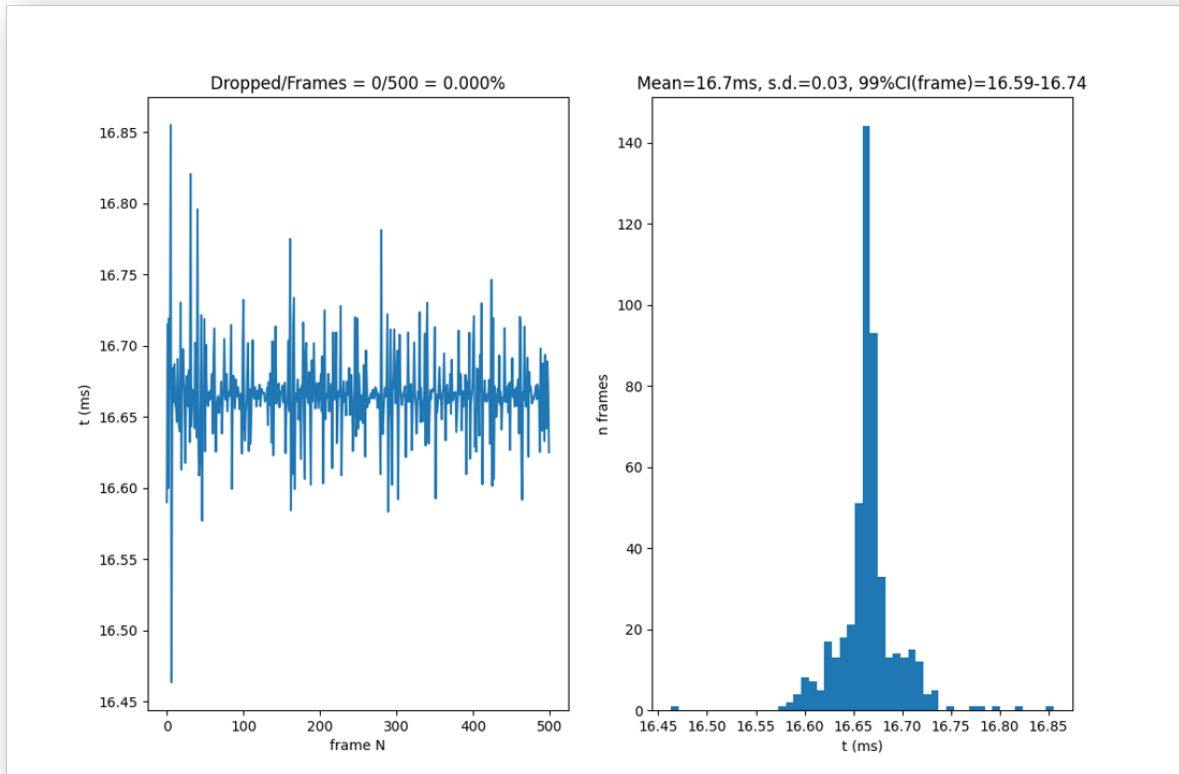


Fig. 2.4: The results here are for a 60Hz monitor, and you can see that there are no dropped frames from the left hand side of the screen, and also the timing of each frame is 16.7ms (shown on the right-hand side of the screen) which is what we would expect from a 60Hz monitor ($1000\text{ms}/60 = 16.66\text{ms}$).

- Use a photodiode or other physical stimulus detector to fully understand the lag, and more importantly the variability of that lag, between any triggers that you send to indicate the start of your stimulus and when the stimulus actually starts.

2.9 Glossary

Adaptive staircase An experimental method whereby the choice of stimulus parameters is not pre-determined but based on previous responses. For example, the difficulty of a task might be varied trial-to-trial based on the participant's responses. These are often used to find psychophysical thresholds. Contrast this with the *method of constants*.

CPU Central Processing Unit is the main processor of your computer. This has a lot to do, so we try to minimise the amount of processing that is needed, especially during a trial, when time is tight to get the stimulus presented on every screen refresh.

CRT Cathode Ray Tube 'Traditional' computer monitor (rather than an LCD or plasma flat screen).

csv Comma-Separated Value files Type of basic text file with 'comma-separated values'. This type of file can be opened with most spreadsheet packages (e.g. MS Excel) for easy reading and manipulation.

GPU Graphics Processing Unit is the processor on your graphics card. The GPUs of modern computers are incredibly powerful and it is by allowing the GPU to do a lot of the work of rendering that is able to achieve good timing precision despite being written in an interpreted language

Method of constants An experimental method whereby the parameters controlling trials are predetermined at the beginning of the experiment, rather than determined on each trial. For example, a stimulus may be presented for 3 pre-determined time periods (100, 200, 300ms) on different trials, and then repeated a number of times. The order of presentation of the different conditions can be randomised or sequential (in a fixed order). Contrast this method with the *adaptive staircase*.

VBI (Vertical Blank Interval, aka the Vertical Retrace, or Vertical Blank, VBL). The period in-between video frames and can be used for synchronising purposes. On a CRT display the screen is black during the VBI and the display beam is returned to the top of the display.

VBI blocking The setting whereby all functions are synced to the VBI. After a call to `psychopy.visual.Window.flip()` nothing else occurs until the VBI has occurred. This is optimal and allows very precise timing, because as soon as the flip has occurred a very precise time interval is known to have occurred.

VBI syncing (aka vsync) The setting whereby the video drawing commands are synced to the VBI. When `psychopy.visual.Window.flip()` is called, the current back buffer (where drawing commands are being executed) will be held and drawn on the next VBI. This does not necessarily entail *VBI blocking* (because the system may return and continue executing commands) but does guarantee a fixed interval between frames being drawn.

xlsx Excel OpenXML file format. A spreadsheet data format developed by Microsoft but with an open (published) format. This is the native file format for Excel (2007 or later) and can be opened by most modern spreadsheet applications including OpenOffice (3.0+), google docs, Apple iWork 08.

INSTALLATION

3.1 Download

For the easiest installation download and install the Standalone package.

For all versions see the [PsychoPy releases on github](#)

is distributed under the [GPL3 license](#)

3.2 Manual installations

See below for options if you don't want to use the Standalone releases:

- *pip install*
- *brew install*
- *Linux*
- *Anaconda and Miniconda*
- *Developers install*

3.2.1 pip install

Now that most python libraries can be installed using *pip* it's relatively easy to manually install and all it's dependencies to your own installation of Python.

The steps are to fetch Python. This method should work on a range of versions of Python but **we strongly recommend you use Python 3.8**. Older Python versions are no longer being tested and may not work correctly. Newer Python versions may not have wheels for all the necessary dependencies even we believe that PsychoPy's code, itself, is compatible all the way up to Python 3.10.

You can install and its dependencies (more than you'll strictly need) by:

```
pip install psychopy
```

If you prefer *not* to install *all* the dependencies (e.g. because the platform or Python version you're on doesn't have that dependency easily available) then you could do:

```
pip install psychopy --no-deps
```

and then install them manually. On Windows, if you need a package that isn't available on PyPI you may want to try the [unofficial packages](#) by Christoph Gohlke

3.2.2 brew install

On a MacOS machine, *brew* can be used to install :

```
brew install --cask psychopy
```

3.2.3 Linux

There used to be neurodebian and Gentoo packages for but these are both badly outdated. We'd recommend you do:

```
# with --no-deps flag if you want to install dependencies manually
pip install psychopy
```

Then fetch a wxPython wheel for your platform from:

<https://extras.wxpython.org/wxPython4/extras/linux/gtk3/>

and having downloaded the right wheel you can then install it with something like:

```
pip install path/to/your/wxpython.whl
```

wxPython>4.0 and doesn't have universal wheels yet which is why you have to find and install the correct wheel for your particular flavor of linux.

Building Python PsychToolbox bindings:

The PsychToolbox bindings for Python provide superior timing for sounds and keyboard responses. Unfortunately we haven't been able to build universal wheels for these yet so you may have to build the pkg yourself. That should not be hard. You need the necessary dev libraries installed first:

```
sudo apt-get install libusb-1.0-0-dev portaudio19-dev libasound2-dev
```

and then you should be able to install using pip and it will build the extensions as needed:

```
pip install psychtoolbox
```

3.2.4 Anaconda and Miniconda

We provide an [environment file](#) that can be used to install and its dependencies. Download the file, open your terminal, navigate to the directory you saved the file to, and run:

```
conda env create -n psychopy -f psychopy-env.yml
```

This will create an environment named `psychopy`. On Linux, the wxPython dependency of is linked against `webkitgtk`, which needs to be installed manually, e.g. via `sudo apt install libwebkitgtk-1.0` on Debian-based systems like Ubuntu.

To activate the newly-created environment and run , execute:

```
conda activate psychopy
psychopy
```

3.2.5 Developers install

Ensure you have Python 3.6 and the latest version of pip installed:

```
python --version
pip --version
```

Next, follow the *instructions to fork and fetch* the latest version of the repository.

From the directory where you cloned the latest repository (i.e., where setup.py resides), run:

```
pip install -e .
```

This will install all dependencies to your default Python distribution (which should be Python 3.6). Next, you should create a new shortcut linking your newly installed dependencies to your current version of in the cloned repository. To do this, simply create a new .BAT file containing:

```
"C:\PATH_TO_PYTHON3.6\python.exe C:\PATH_TO_CLONED_PSYCHOPY_REPO\psychopy\app\  
↪psychopyApp.py"
```

Alternatively, you can run the psychopyApp.py from the command line:

```
python C:\PATH_TO_CLONED_PSYCHOPY_REPO\psychopy\app\psychopyApp
```

3.3 Recommended hardware

The minimum requirement for is a computer with a graphics card that supports OpenGL. Many newer graphics cards will work well. Ideally the graphics card should support OpenGL version 2.0 or higher. Certain visual functions run much faster if OpenGL 2.0 is available, and some require it (e.g. ElementArrayStim).

If you already have a computer, you can install and the Configuration Wizard will auto-detect the card and drivers, and provide more information. It is inexpensive to upgrade most desktop computers to an adequate graphics card. High-end graphics cards can be very expensive but are only needed for very intensive use.

Generally NVIDIA and ATI (AMD) graphics chips have higher performance than Intel graphics chips so try and get one of those instead.

3.3.1 Notes on OpenGL drivers

On Windows, if you get an error saying “**pyglet.gl.ContextException: Unable to share contexts**” then the most likely cause is that you need OpenGL drivers and your built-in Windows only has limited support for OpenGL (or possibly you have an Intel graphics card that isn’t very good). Try installing new drivers for your graphics card **from its manufacturer’s web page**, not from Microsoft. For example, [NVIDIA provides drivers for its cards here](#)

GETTING STARTED

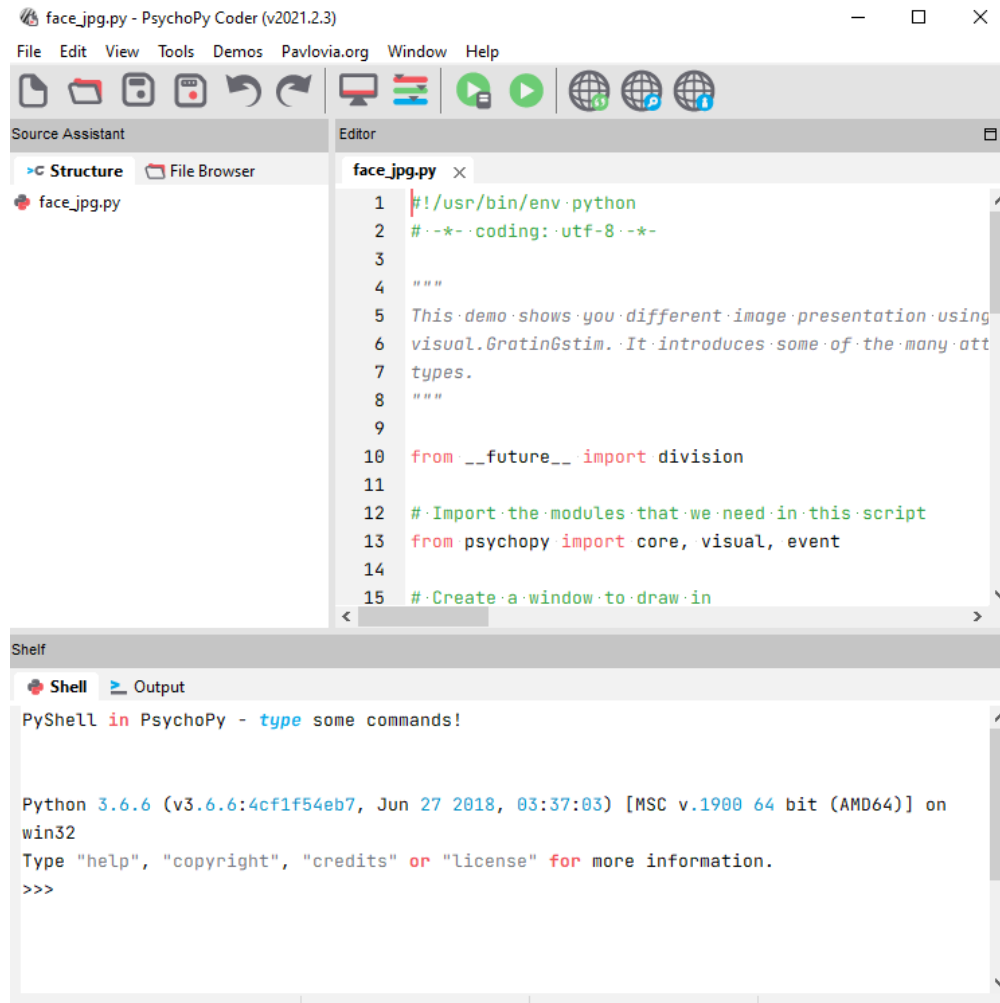
As an application, has two main views: the *Builder* view, and the *Coder* view. It also has a underlying *Reference Manual (API)* that you can call directly.

1. *Builder*. You can generate a wide range of experiments easily from the Builder using its intuitive, graphical user interface (GUI). This might be all you ever need to do. But you can always compile your experiment into a python script for fine-tuning, and this is a quick way for experienced programmers to explore some of PsychoPy's libraries and conventions. **Note: if you are taking a study online we highly advise even experienced coders use Builder view, as the JS version of your experiment will also be generated**



1. *Coder*. For those comfortable with programming, the Coder view provides a basic code editor with syntax highlighting, code folding, and so on. Importantly, it has its own output window and Demo menu. The demos illustrate how to do specific tasks or use specific features; they are not whole experiments. The *Coder tutorials* should help get you going, and the *Reference Manual (API)* will give you the details.

The Builder and Coder views are the two main aspects of the application. If you've installed the StandAlone version of on **MS Windows** then there should be an obvious link to in your > Start > Programs. If you installed the StandAlone version on **macOS** then the application is where you put it (!). On these two platforms you can open the Builder and Coder views from the View menu and the default view can be set from the preferences. **On Linux**, you can start from a command line, or make a launch icon (which can depend on the desktop and distro). If the app is started with flags —coder (or -c), or —builder (or -b), then the preferences will be overridden and that view will be created as the app opens.



For experienced python programmers, it's possible to use without ever opening the Builder or Coder. Install the libraries and dependencies, and use your favorite IDE instead of the Coder.

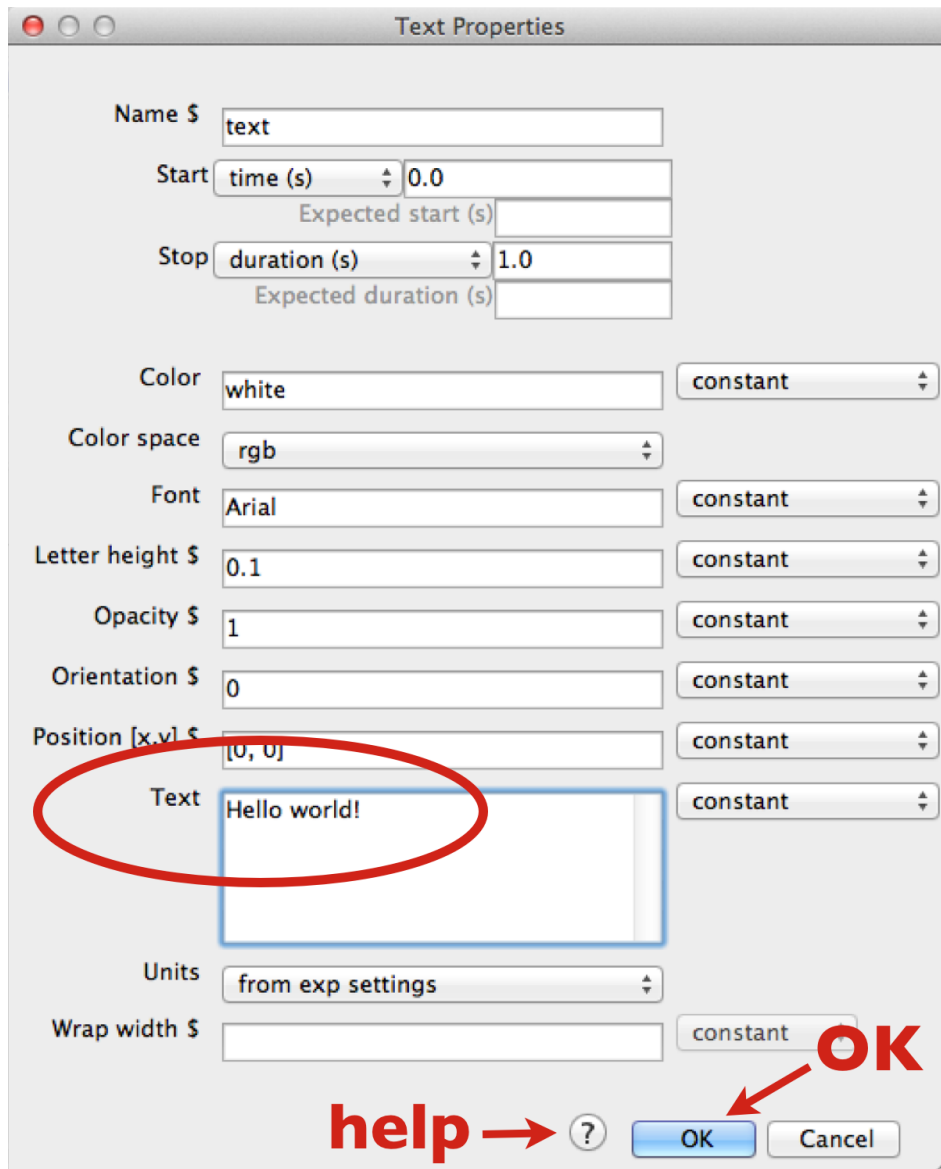
4.1 Builder

When learning a new computer language, the classic first program is simply to print or display “Hello world!”. Lets do it.

4.1.1 A first program

Start , and be sure to be in the Builder view.

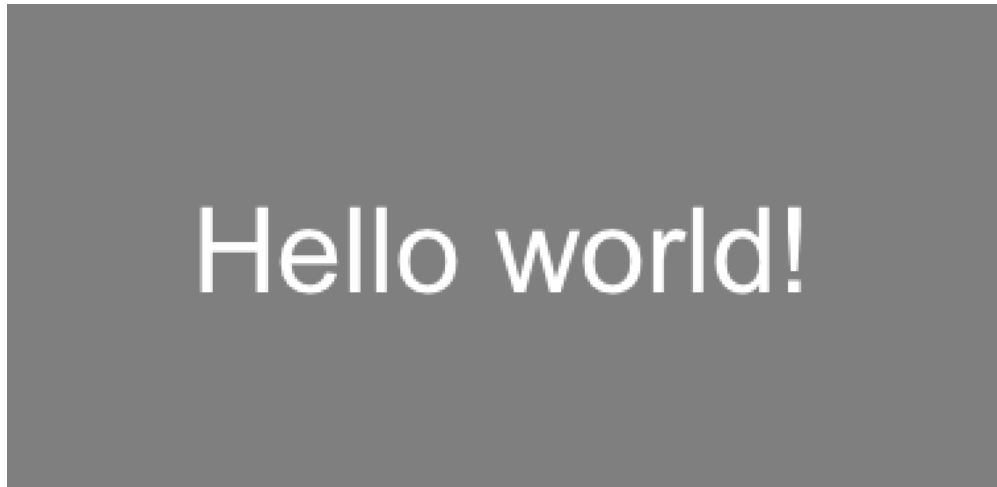
- If you have poked around a bit in the Builder already, be sure to start with a clean slate. To get a new Builder view, type *Ctrl-N* on Windows or Linux, or *Cmd-N* on Mac.
- Click on a Text component and a Text Properties dialog will pop up.



- In the *Text* field, replace the default text with your message. When you run the program, the text you type here will be shown on the screen.
- Click OK (near the bottom of the dialog box). (Properties dialogs have a link to online help—an icon at the bottom, near the OK button.)
- Your text component now resides in a routine called *trial*. You can click on it to view or edit it. (Components, Routines, and other Builder concepts are explained in the *Builder documentation*.)
- Back in the main Builder, type *Ctrl-R* (Windows, Linux) or *Cmd-R* (Mac), or use the mouse to click the *Run* icon.



Assuming you typed in “Hello world!”, your screen should have looked like this (briefly):



If nothing happens or it looks wrong, recheck all the steps above; be sure to start from a new Builder view.

What if you wanted to display your cheerful greeting for longer than the default time?

- Click on your Text component (the existing one, not a new one).
- Edit the *Stop duration (s)* to be 3.2; times are in seconds.
- Click OK.
- And finally *Run*.

When running an experiment, you can quit by pressing the *escape* key (this can be configured or disabled). You can quit from the File menu, or typing *Ctrl-Q / Cmd-Q*.

4.1.2 Getting beyond Hello

To do more, you can try things out and see what happens. You may want to consult the *Builder documentation*. Many people find it helpful to explore the Builder demos, in part to see what is possible, and especially to see how different things are done.

A good way to develop your own first experiment is to base it on the Builder demo that seems closest. Copy it, and then adapt it step by step to become more and more like the program you have in mind. Being familiar with the Builder demos can only help this process.

You could stop here, and just use the Builder for creating your experiments. It provides a lot of the key features that people need to run a wide variety of studies. But it does have its limitations. When you want to have more complex designs or features, you’ll want to investigate the Coder. As a segue to the Coder, let’s start from the Builder, and see how Builder programs work.

4.2 Builder-to-coder

Whenever you run a Builder experiment, will first translate it into python code, and then execute that code.

To get a better feel for what was happening “behind the scenes” in the Builder program above:

- In the Builder, load or recreate your “hello world” program.
- Instead of running the program, explicitly convert it into python: Type *F5*, or click the *Compile* icon:



The view will automatically switch to the Coder, and display the python code. If you then save and run this code, it would look the same as running it directly from the Builder.

It is always possible to go from the Builder to python code in this way. You can then edit that code and run it as a python program. However, you **cannot go from code back to a Builder representation** editing in coder is a one-way street, so, in general, we advise compiling to code is good for understanding what exists but, where possible, make code tweaks in builder itself using code components.

To switch quickly between Builder and Coder views, you can type *Ctrl-L / Cmd-L*.

4.3 Coder

Being able to inspect Builder-generated code is nice, but it’s possible to write code yourself, directly. With the Coder and various libraries, you can do virtually anything that your computer is capable of doing, using a full-featured modern programming language (python).

For variety, lets say hello to the Spanish-speaking world. knows Unicode (UTF-8).

If you are not in the Coder, switch to it now.

- Start a new code document: *Ctrl-N / Cmd-N*.
- Type (or copy & paste) the following:

```
from psychopy import visual, core

win = visual.Window()
msg = visual.TextStim(win, text=u"\u00A1Hola mundo!")

msg.draw()
win.flip()
core.wait(1)
win.close()
```

- Save the file (the same way as in Builder).
- Run the script.

Note that the same events happen on-screen with this code version, despite the code being much simpler than the code generated by the Builder. (The Builder actually does more, such as prompt for a subject number.)

Coder Shell

The shell provides an interactive python interpreter, which means you can enter commands here to try them out. This provides yet another way to send your salutations to the world. By default, the Coder's output window is shown at the bottom of the Coder window. Click on the Shell tab, and you should see python's interactive prompt, >>>:

```
PyShell in |PsychoPy| - type some commands!  
  
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

At the prompt, type:

```
>>> print(u"\u00A1Hola mundo!")
```

You can do more complex things, such as type in each line from the Coder example directly into the Shell window, doing so line by line:

```
>>> from psychopy import visual, core
```

and then:

```
>>> win = visual.Window()
```

and so on—watch what happens each line:

```
>>> msg = visual.TextStim(win, text=u"\u00A1Hola mundo!")  
>>> msg.draw()  
>>> win.flip()
```

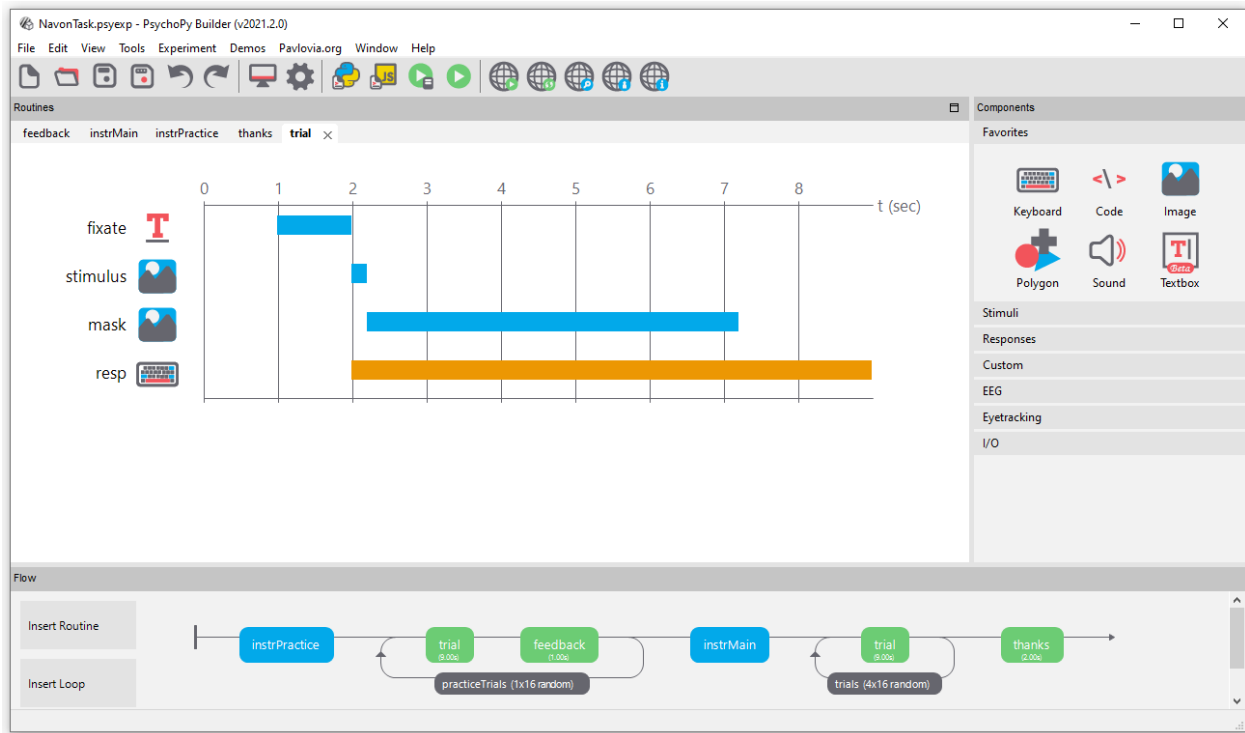
and so on. This lets you try things out and see what happens line-by-line (which is how python goes through your program).

5.1 Building experiments in a GUI

Making your experiments using the builder is the approach that we generally recommend. Why would we (a team of programmers) recommend using a GUI?:

- It's much faster to make experiments
- Your experiment will be less likely to have bugs (experiments coded from scratch can very easily contain errors - even when made by the best of programmers!).
- You can easily make an experiment to run **online in a browser**. builder view is writing you a python script "under the hood" of your experiment, but if you want to run an experiment online it can also compile a javascript version of your task using PsychoPy's sister library [PsychoJS](#). Remember that PsychoJS is younger than - so remember to check the [status of online options](#) *before* making an experiment you plan to run online! The easiest way to host a study online from is through the platform, and builder has inbuilt integration to interact with this platform.

There are a number of tutorials on how to get started making experiments in builder on the [PsychoPy Youtube channel](#) as well as several written tutorials and [Experiment Recipes](#). You can also find a range of [materials for teaching](#) using builder view.



Contents:

5.1.1 Builder concepts

Routines and Flow

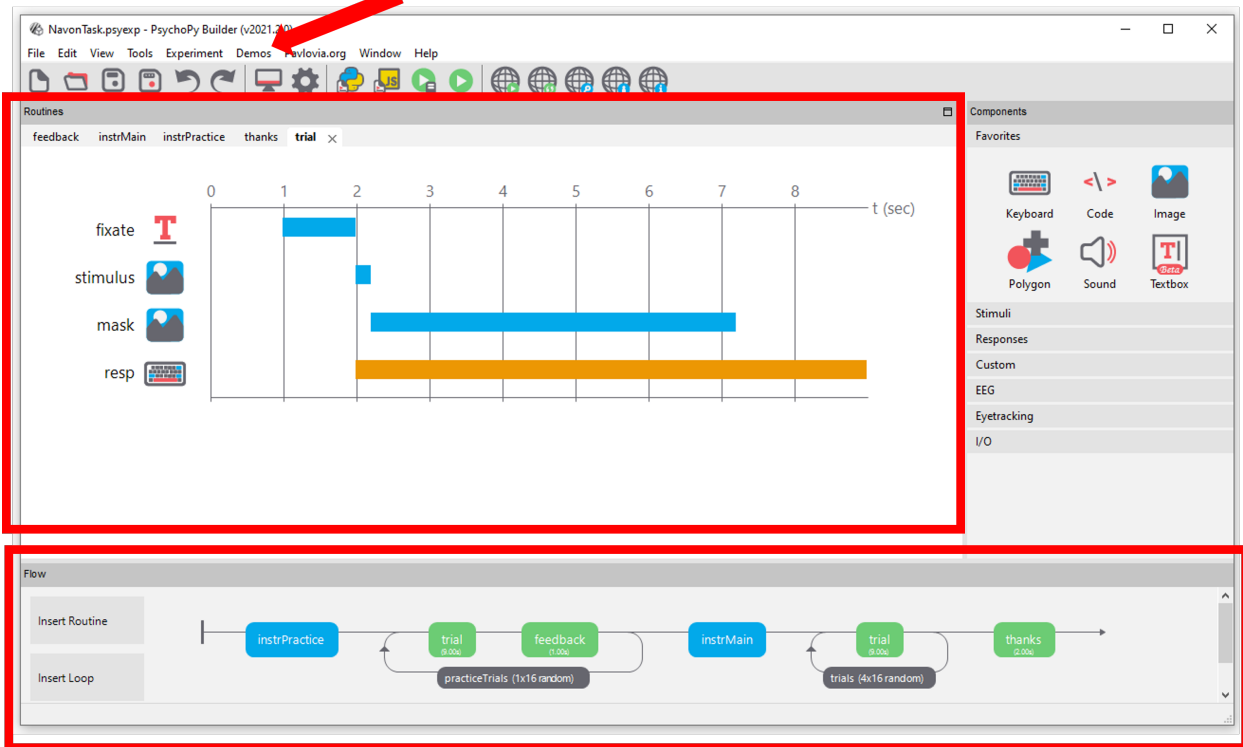
The Builder view of the application is designed to allow the rapid development of a wide range of experiments for experimental psychology and cognitive neuroscience experiments.

The Builder view comprises two main panels for viewing the experiment's *Routines* (upper left) and another for viewing the *Flow* (lower part of the window).

An experiment can have any number of *Routines*, describing the timing of stimuli, instructions and responses. These are portrayed in a simple track-based view, similar to that of video-editing software, which allows stimuli to come on off repeatedly and to overlap with each other.

The way in which these *Routines* are combined and/or repeated is controlled by the *Flow* panel. All experiments have exactly one *Flow*. This takes the form of a standard flowchart allowing a sequence of routines to occur one after another, and for loops to be inserted around one or more of the *Routines*. The loop also controls variables that change between repetitions, such as stimulus attributes.

If it is your first time opening , we highly recommend taking a look at the large number of inbuilt demos that come with . This can be done through selecting *Demos > unpack demos* within your application. Another good place to get started is to take a look at the many [openly available demos at pavlovia.org](#) you can view an [intro to Pavlovia](#) at our Youtube channel.



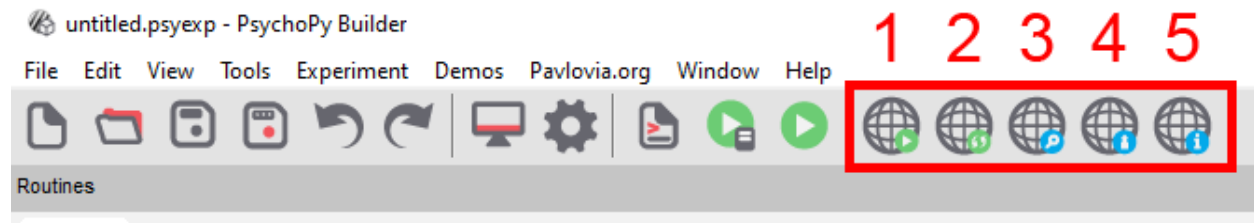
The PsychoPy builder, the Routines panel and the Flow are highlighted, if you are new to PsychoPy, we recommend starting by unpacking your demos and exploring the example tasks

The components panel

You can add components to an experiment by selecting components from the *Components panel*. This is currently divided into 7 sections:

- *Favorites* - your commonly used components
- *Stimuli* - components used to present a stimulus (e.g. a visual image or shape, or an auditory tone or file)
- *Responses* - stimuli used to gather responses (e.g. keyboards or mouse components - amongst many others!)
- *Custom* - builder can be used to make a fair few complex experiments now, but for added flexibility, you can add code components at any point in an experiment (e.g. for providing response-dependant feedback).
- *EEG* - can actually be used with a range of EEG devices. Most of these are interacted with through delivering a trigger through the parallel port (see *I/O* below), or serial port (see ../api/serial.html). However, Builder has inbuilt support (i.e. no need for code snippets) for working with Emotiv EEG, you can view a [Youtube tutorial on how to use Emotiv EEG with PsychoPy here](#).
- *Eyetracking* - 2021.2 released inbuilt support for eyetrackers! had supported eye tracker research for a while, but not via components in builder. You can learn more about these from the [more specific components.html](#) info.
- *I/O* - I/O stands for “input/output” under the hood this is ../api/iohub.html, this is useful for if you are working with external hardware devices requiring communication via the parallel port (e.g. EEG).

Making experiments to go online



Buttons to interact with pavlovian.org from your experiment builder

Before making an experiment to go online, it is a good idea to check the [status of online options](#) - remember PsychoJS (the javascript sister library of) is younger than - so not everything can be done online yet! but for most components there are prototype work arounds to still make things possible (e.e. RDKs and staircases). You can learn more about taking experiments online from builder [via the online documentation](#).

5.1.2 Routines

An experiment consists of one or more Routines. A Routine might specify the timing of events within a trial or the presentation of instructions or feedback. Multiple Routines can then be combined in the *Flow*, which controls the order in which these occur and the way in which they repeat.

To create a new Routine, you can either select “Insert Routine” from within your flow panel or use the Experiment menu. The display size of items within a routine can be adjusted (see the View menu).

Within a Routine there are a number of components. These components determine the occurrence of a stimulus, or the recording of a response. Any number of components can be added to a Routine. Each has its own line in the Routine view that shows when the component starts and finishes in time, and these can overlap.

For now the time axis of the Routines panel is fixed, representing seconds (one line is one second). If you choose to present your stimuli based on another timing unit, e.g. number of frames (more precise) and can be scaled up or down to allow you can specify the “Expected duration” within the component - that will mean that this component still appears on your routine timeline.

5.1.3 Flow

In the Flow panel a number of *Routines* can be combined to form an experiment. For instance, your study may have a *Routines* that presented initial instructions and waited for a key to be pressed, followed by a *Routines* that presented one trial which should be repeated 5 times with various different parameters set. All of this is achieved in the Flow panel. You can adjust the display size of the Flow panel (see View menu).

Adding Routines

The *Routines* that the Flow will use should be generated first (although their contents can be added or altered at any time). To insert a *Routines* into the Flow click the appropriate button in the left of the Flow panel or use the Experiment menu. A dialog box will appear asking which of your *Routines* you wish to add. To select the location move the mouse to the section of the flow where you wish to add it and click on the black disk.

Loops

Loops control the repetition of *Routines* and the choice of stimulus parameters for each. To insert a loop use the button on the left of the Flow panel, or the item in the Experiment menu of the Builder. The start and end of a loop is set in the same way as the location of a *Routines* (see above). Loops can encompass one or more *Routines* and other loops (i.e. they can be nested).

As with components in *Routines*, the loop must be given a name, which must be unique and made up of only alphanumeric characters (underscores are allowed). I would normally use a plural name, since the loop represents multiple repeats of something. For example, *trials*, *blocks* or *epochs* would be good names for your loops.

It is usually best to use trial information that is contained in an external file (.xlsx or .csv). When inserting a *loop* into the *flow* you can browse to find the file you wish to use for this. An example of this kind of file can be found in the Stroop demo (trialTypes.xlsx). The column names are turned into variables (in this case text, letterColor, corrAns and congruent), these can be used to define parameters in the loop by putting a \$ sign before them e.g. \$text.

As the column names from the input file are used in this way they must have legal variable names i.e. they must be unique, have no punctuation or spaces (underscores are ok) and must not start with a digit.

The parameter *Is trials* exists because some loops are not there to indicate trials *per se* but a set of stimuli within a trial, or a set of blocks. In these cases we don't want the data file to add an extra line with each pass around the loop. This parameter can be unchecked to improve (hopefully) your data file outputs. [Added in v1.81.00]

Loop types

You can use a number of different “Loop Types” in , this controls the way in which the trials you have fed into the “Conditions” field are presented. Imagine you have a conditions file that looks like this:

```
letter
a
b
c
```

After saving this as a spreadsheet (.xlsx or .csv), we could then add this to the “Conditions” field of our loop. Let's imagine we want to present each letter twice, so we set *nReps* to 2. We could then use the following Loop Types:

- **Random** - present a - b in a random order, because we have *nReps* at 2, this would be repeated twice e.g. [c, a, b, a, c, b]
- **Full Random** - present a - b in a random order but also take into account the number of *nReps*. Here, imagine that rather than having 3 items in the bag that we sample from, and repeat this twice, we instead have 6 items in the bag that are randomly sampled from. This would mean that with *fullRandom*, but not *random*, it would be possible to get the following order of trials e.g. [a, a, b, c, c, b] - notice that a was sampled twice in the first 2 trials.
- **sequential** - present the rows in the order they are set in the spreadsheet. Currently does not have inbuilt support for specific randomisation constraints, so if you need a specific pseudorandom order, preset this in your spreadsheet file and use a “sequential” loopType.
- **staircase** - for use with adaptive procedures, create an output variable called `level` that can then be used to set the parameter of a stimulus (e.g. its opacity) in an adaptive fashion. This allows researchers to converge upon a participants threshold by adjusting the value of `level` in accordance with performance.
- **interleaved staircases** - for use with multiple staircases that are interleaved. This can also be used to implement other staircasing algorithms such as [QUEST \(Watson and Pelli, 1983\)](#) via `QuestHandler`.

Selecting a subset of conditions

In the standard *Loop types* you would use all the rows/conditions within your conditions file. However there are often times when you want to select a subset of your trials before randomising and repeating.

The parameter *Select rows* allows this. You can specify which rows you want to use by inserting values here:

- *0,2,5* gives the 1st, 3rd and 6th entry of a list - Python starts with index zero)
- *\$random(4)*10* gives 4 indices from 0 to 9 (so selects 4 out of 10 conditions)
- *5:10* selects the 6th to 10th rows
- *\$myIndices* uses a variable that you've already created

Note in the last case that *5:8* isn't valid syntax for a variable so you cannot do:

```
myIndices = 5:8
```

but you can do:

```
myIndices = slice(5,8) #python object to represent a slice
myIndices = "5:8" #a string that PsychoPy can then parse as a slice later
myIndices = "5:8:2" #as above but
```

Note that uses Python's built-in slicing syntax (where the first index is zero and the last entry of a slice doesn't get included). You might want to check the outputs of your selection in the Python shell (bottom of the Coder view) like this:

```
>>> range(100)[5:8] #slice 5:8 of a standard set of indices
[5, 6, 7]
>>> range(100)[5:10:2] #slice 5:8 of a standard set of indices
[5, 7, 9, 11, 13, 15, 17, 19]
```

Check that the conditions you wanted to select are the ones you intended!

Using loops to update stimuli trial-by-trial

Once you have a loop around the routine you want to repeat, you can use the variables created in your conditions file to update any parameter within your routine. For example, let's say that you have a conditions file that looks like this:

```
letter
a
b
c
```

You could then add a Text component and in the *text* field type *\$letter* and then set the corresponding dropdown box to "set every repeat". This indicates that you want the value of this parameter to change on each iteration of your loop, and the value of that parameter on each loop will correspond to the value of "letter" drawn on each trial.

Note: You only need to use the \$ sign if that field name does not already contain a \$ sign! You also don't need several dollar signs in a field e.g. you wouldn't set the position of a stimulus on each repeat using (*\$myX*, *\$myY*) instead you would just use *\$(myX, myY)* - this is because the dollar sign indicates that this field will now accept python code, rather than that this value corresponds to a variable.

5.1.4 Blocks of trials and counterbalancing

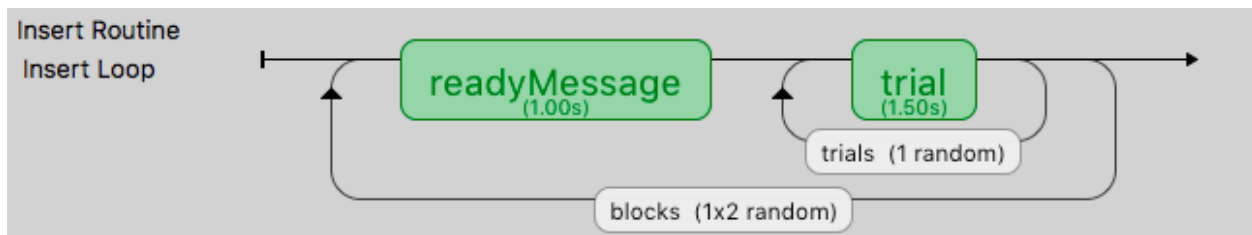
Many people ask how to create blocks of trials, how to randomise them, and how to counterbalance their order. This isn't all that hard, although it does require a bit of thinking!

Blocking similar conditions

The key thing to understand is that you should not create different Routines for different trials in your blocks (if at all possible). Try to define your trials with a single Routine. For instance, let's imagine you're trying to create an experiment that presents a block of pictures of houses or a block of faces. It would be tempting to create a Routine called *presentFace* and another called *presentHouse* but you actually want just one called *presentStim* (or just *trial*) and then set that to differ as needed across different stimuli.

This example is included in the Builder demos, as of 1.85, as "images_blocks".

You can add a loop around your trials, as normal, to control the trials within a block (e.g. randomly selecting a number of images) but then you will have a second loop around this to define how the blocks change. You can also have additional Routines like something to inform participants that the next block is about to start.



So, how do you get the block to change from one set of images to another? To do this create three spreadsheets, one for each block, determining the filenames within that block, and then another to control which block is being used:

faceBlock.xlsx		houseBlock.xlsx		
	A	B		
1	stimFile		1	stimFile
2	stims/face01.jpg		2	stims/house01.jpg
3	stims/face02.jpg		3	stims/house02.jpg
4	stims/face03.jpg		4	stims/house03.jpg

chooseBlocks.xlsx		
	A	B
1	condsFile	readyMsg
2	facesBlock.csv	Some faces
3	housesBlock.csv	Some houses

Setting up the basic conditions. The facesBlock, and housesBlock, files look more like your usual conditions files. In this example we can just use a variable *stimFile* with values like *stims/face01.jpg* and *stims/face02.jpg* while the housesBlock file has *stims/house01.jpg* and *stims/house02.jpg*. In a real experiment you'd probably also have response keys and suchlike as well.

So, how to switch between these files? That's the trick and that's what the other file is used for. In the *chooseBlocks.xlsx* file you set up a variable called something like *condsFile* and that has values of *facesBlock.xlsx* and *housesBlock.xlsx*. In the outer (blocks) loop you set up the conditions file to be *chooseBlocks.xlsx* which creates a variable *condsFile*. Then, in the inner (trials) loop you set the conditions file not to be any file directly but simply *\$condsFile*. Now, when starts this loop it will find the current value of *condsFile* and insert the appropriate thing, which will be the name of an conditions file and we're away!

Your *chooseBlocks.xlsx* can contain other values as well, such as useful identifiers. In this demo you could add a value *readyText* that says "Ready for some houses", and "Ready for some faces" and use this in your get ready Routine.

Variables that are defined in the loops are available anywhere within those. In this case, of course, the values in the outer loop are changing less often than the values in the inner loop.

Counterbalancing similar conditions

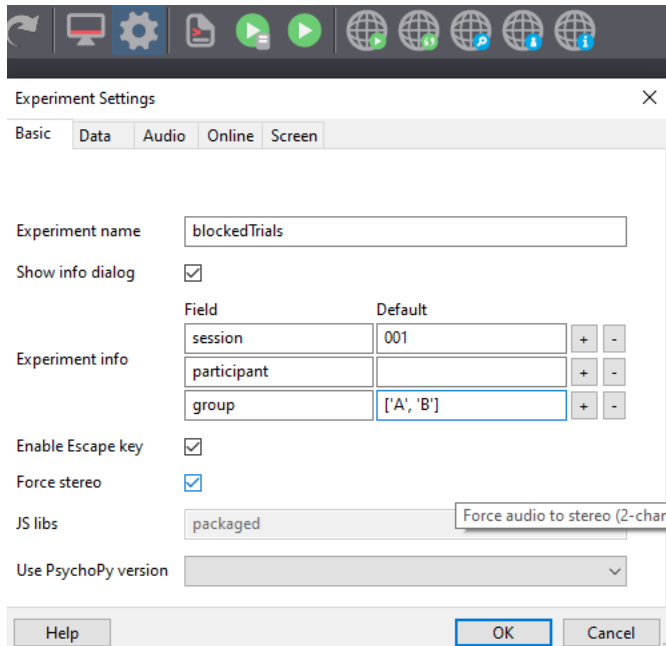
Counterbalancing is simply an extension of blocking. Until now, we have a *randomised block design*, where the order of blocks is set to random. At the moment we also only have one repeat of each block, but we could also present more than one repeat of each block by controlling the number of rows assigned to each block in our ‘chooseBlocks’ file.

In a counterbalanced design you want to control the order explicitly and you want to provide a different order for different groups of participants. Maybe group A always gets faces first, then houses, and group B always gets houses first, then faces.

Now we need to create further conditions files, to specify the exact orders we want, so we’d have something like *chooseBlockA.xlsx* and *chooseBlockB.xlsx*:

chooseBlocksA.xlsx			chooseBlocksB.xlsx		
	A	B		A	B
1	condsFile	readyMsg	1	condsFile	readyMsg
2	facesBlock.csv	Some faces	2	housesBlock.csv	Some houses
3	housesBlock.csv	Some houses	3	facesBlock.csv	Some faces

The last part of the puzzle is how to assign participants to groups. For this you *could* write a Code Component that would generate a variable for you (if...: `groupFile = "groupB.xlsx"`) but the easiest thing is probably that you, the experimenter, chooses this using the GUI we present at the start of the experiment. So, we add a field to our GUI using experiment settings:



Note that entering a *list* as the default input will present us with a dropdown in our GUI.

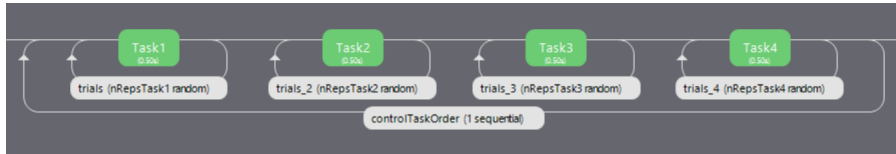
Finally, we set parameters of our *blocks* loop to use the method ‘sequential’ (because we are using a predefined order) and we enter the following into the conditions field: `~ $"chooseBlocks"+expInfo['group']+".xlsx" ~` This will concatenate the string “chooseBlocks” with our selected group (“A” or “B”) and the required file extension (in this case “xlsx”) in order to select the correct order.

Even though our outer loop is now sequential, your inner loop still probably wants to be random (to shuffle the image order within a block).

Counterbalancing different subtasks

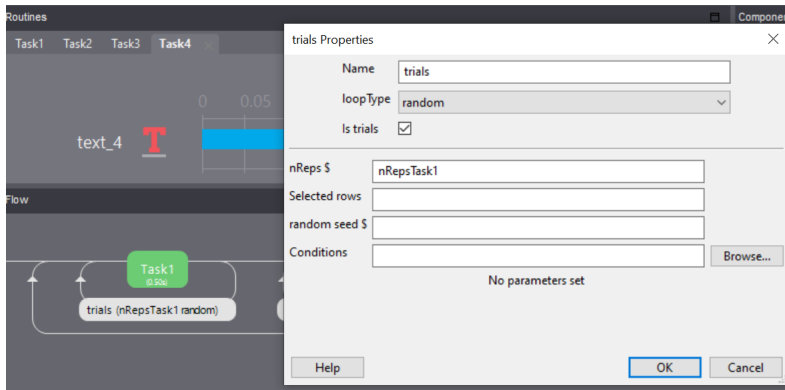
The above example is useful when we have multiple blocks where the routines we present would be largely similar (i.e. both blocks present an image component), but what about situations where we have totally different tasks we need to counterbalance (e.g. an auditory stroop and an n-back task). The following method is an extension of the logic used in the ‘branchedExp’ demo available in builderview. You can download a [working version of the example](#) we will work through.

So, imagine we have 4 very different tasks. Our flow might look something like this:



Here we have 4 totally different tasks, each with its own loop. Now imagine one participant is presented with these tasks using the order Task1 -> Task2 -> Task3 -> Task 4 (for ease let’s call this group, ‘ABCD’) whilst another is presented with Task2 -> Task3 -> Task4 -> Task 1 (let’s call them group ‘BCDA’).

The loop surrounding each task will look something like this (although here I have stripped the parameters to the bare minimum, you will likely have a conditions file):



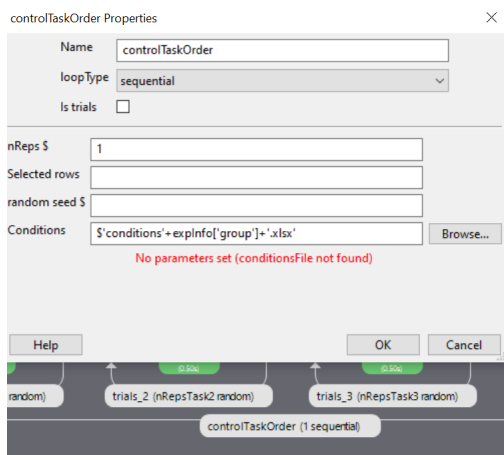
Where the number of times that block is repeated (or occurs at all!) is determined by the outer loop (e.g. Task1 nReps = ‘nRepsTask1’, Task2 nReps = ‘nRepsTask2’ and so on).

For our outer loop we will use conditions files that look something like this:

	A	B	C	D	E
nRepsTask1		nRepsTask2	nRepsTask3	nRepsTask4	
	1	0	0	0	
	0	1	0	0	
	0	0	1	0	
	0	0	0	1	

Each row corresponds to how many times a subtask routine (or set of routines) will be repeated per iteration of the outer loop. The example conditions file above would be used for a participant in group ‘ABCD’ (on the first iteration Task 1 will repeat once, on the second iteration Task 2 will repeat once and so on).

Just like before we create a field in our experiment settings called group (but let’s say that the group names this time are ‘ABCD’, ‘BCDA’ and so on where the content of the conditions file differs). Finally, we use the following parameters for our outermost loop to select which, preordered, conditions file we are using.



Using this method, we could present several subtasks in a counterbalanced order (without having to create new experiment files for each order!).

What about going **online** ? Well, things are more difficult there, but not impossible let's talk about *Counterbalancing online*

5.1.5 Components

Routines in the Builder contain any number of components, which typically define the parameters of a stimulus or an input/output device.

The following components are available, as at version 1.65, but further components will be added in the future including Parallel/Serial ports and other visual stimuli (e.g. GeometricStim).

Aperture Component

This component can be used to filter the visual display, as if the subject is looking at it through an opening (i.e. add an image component, as the background image, then add an aperture to show part of the image). Currently, in builder, only circular apertures are supported (you can change the shape by specifying your aperture in a code component- we are hoping to make it easier to do this through builder soon!). Moreover, only one aperture is enabled at a time. You can't "double up": a second aperture takes precedence. Currently this component **does not run online** (see the status of online options, but you can achieve something similar online using an image with a mask: see an [example demo here](#) with corresponding [PsychoPy experiment files here](#) or by using the [MouseView](#) plugin.

Basic

name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the aperture should start having its effect. See *Defining the onset/duration of components* for details.

expected start(s) : If you are using frames to control timing of your stimuli, you can add an expected start time to display the component timeline in the routine.

stop : When the aperture stops having its effect. See *Defining the onset/duration of components* for details.

expected duration(s) : If you are using frames to control timing of your stimuli, you can add an expected duration to display the component timeline in the routine.

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

size [integer] The size controls how big the aperture will be, in pixels, default = 120

pos [[X,Y]] The position of the centre of the aperture, in the units specified by the stimulus or window.

Note: Top tip: You can make an aperture (or anything!) track the position of your mouse by adding a mouse component, then setting the position of your aperture to be `mouse.getPos()` (and *set every frame*), where “mouse” corresponds to the name of your mouse component.

spatial units : What units to use.

See also:

API reference for *Aperture*

Brush Component

The Brush component is a freehand drawing tool.

Properties

Name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start [int, float] The time that the stimulus should first appear.

Stop [int, float] Governs the duration for which the stimulus is presented.

Press Button [bool] Should the participant have to press a button to paint, or should it be always on?

Appearance

How should the stimulus look? Colour, borders, etc.

Brush Size [int, float] Width of the line drawn by the brush, in pixels

Opacity : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

Brush Color [color] Colour of the brush

Brush Color Space [rgb, dkl, lms, hsv] See *Color spaces*

See also:

API reference for *Brush*

Button Component

This component allows you to show a static textbox which ends the routine and/or triggers a “callback” (some custom code) when pressed. The nice thing about the button component is that you can allow mouse/touch responses with a single component instead of needing 3 separate components i.e. a textbox component (to display as a “clickable” thing), a mouse component (to click the textbox) and a code component (not essential, but for example to check if a clicked response was correct or incorrect).

name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : The duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

Force End Routine on Press If this box is checked then the *Routine* will end as soon as one of the mouse buttons is pressed.

button text [string] Text to be shown

callback function [code] Custom code to run when the button is pressed

run once per click [bool] Whether the callback function to only run once when the button is initially clicked, or whether it should run continuously each frame while the button is pressed.

Appearance

How should the stimulus look? Colour, borders, etc.

text color [color] See *Color spaces*

fill color [color] See *Color spaces*

border color [color] See *Color spaces*

color space [rgb, dkl, lms, hsv] See *Color spaces*

border width [int | float] How wide should the line be? Width is specified in chosen spatial units, see *Units for the window and stimuli*

opacity : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

ori [degrees] The orientation of the stimulus in degrees.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

size [(width, height)] Size of the stimulus on screen

spatial units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

padding [float] How much space should there be between the box edge and the text?

anchor [center, center-left, center-right, top-left, top-center, top-right, bottom-left, bottom-center, bottom-right] What point on the button should be anchored to its position? For example, if the position of the button is (0, 0), should the middle of the button be in the middle of the screen, should its top left corner be in the middle of the screen, etc.?

Formatting

Formatting text

font [string] What font should the text be set in? Can be a font installed on your computer, saved to the “fonts” folder in your user folder or (if you are connected to the internet), a font from Google Fonts.

language style [LTR, RTL, Arabic] Should text be laid out from left to right (LTR), from right to left (RTL), or laid out like Arabic script?

letter height [integer or float] The height of the characters in the given units of the stimulus/window. Note that nearly all actual letters will occupy a smaller space than this, depending on font, character, presence of accents etc. The width of the letters is determined by the aspect ratio of the font.

line spacing [float] How tall should each line be, proportional to the size of the font?

See also:

API reference for `ButtonStim`

Camera Component

The camera component provides a way to use the webcam to record participants during an experiment. **Note: For online experiments, the browser will notify participants to allow use of webcam before the start of the task.**

When recording via webcam, specify the starting time relative to the start of the routine (see *start* below) and a stop time (= duration in seconds). A blank duration evaluates to recording for 0.000s.

The resulting video files are saved in .mp4 format if recorded locally and saved in .webm if recorded online. There will be one file per recording. The files appear in a new folder within the data directory in a folder called `data_cam_recorded`. The file names include the unix (epoch) time of the onset of the recording with milliseconds, e.g., *recording_cam_2022-06-16_14h32.42.064.mp4*. **Note: For online experiments, the recordings can only be downloaded from the “Download results” button from the study’s Pavlovia page.**

For a demo in builder mode, after unpacking the demos, click on Demos > Feature Demos > camera. For a demo in coder mode, click on Demos > hardware > camera.py

Parameters

Basic

name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the stimulus should first play. See *Defining the onset/duration of components* for details.

stop (duration): The length of time (sec) to record for. An *expected duration* can be given for visualisation purposes. See *Defining the onset/duration of components* for details; note that only seconds are allowed.

Data

Save onset/offset times: bool Whether to save the onset and offset times of the component.

Sync timing with screen refresh: bool Whether to sync the start time of the component with the window refresh.

Output File Type: File type the video is saved as locally is mp4 and for online it is webm.

Cedrus Button Box Component

This component allows you to connect to a Cedrus Button Box to collect key presses.

Note that there is a limitation currently that a button box can only be used in a single Routine. Otherwise |PsychoPy| tries to initialise it twice which raises an error. As a workaround, you need to insert the start-routine and each-frame code from the button box into a code component for a second routine.

Properties

Name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the button box is first read. See *Defining the onset/duration of components* for details.

Stop : Governs the duration for which the button box is first read. See *Defining the onset/duration of components* for details.

Force end of Routine [true/false] If this is checked, the first response will end the routine.

Data

What information to save, how to lay it out and when to save it.

Allowed keys [None, or an integer, list, or tuple of integers 0-7] This field lets you specify which buttons (None, or some or all of 0 through 7) to listen to.

Store [(choice of: first, last, all, nothing)] Which button events to save in the data file. Events and the response times are saved, with RT being recorded by the button box (not by).

Store correct [true/false] If selected, a correctness value will be saved in the data file, based on a match with the given correct answer.

Discard previous [true/false] If selected, any previous responses will be ignored (typically this is what you want).

Hardware

Parameters for controlling hardware.

Device number: integer This is only needed if you have multiple Cedrus devices connected and you need to specify which to use.

Use box timer [true/false] Set this to True to use the button box timer for timing information (may give better time resolution)

See also:

API reference for `iolab`

Code Component

The *Code Component* can be used to insert short pieces of python code into your experiments. This might be create a variable that you want for another *Component*, to manipulate images before displaying them, to interact with hardware for which there isn't yet a pre-packaged component in (e.g. writing code to interact with the serial/parallel ports). See *code uses* below.

Be aware that the code for each of the components in your *Routine* are executed in the order they appear on the *Routine* display (from top to bottom). If you want your *Code Component* to alter a variable to be used by another component immediately, then it needs to be above that component in the view. You may want the code not to take effect until next frame however, in which case put it at the bottom of the *Routine*. You can move *Components* up and down the *Routine* by right-clicking on their icons.

Within your code you can use other variables and modules from the script. For example, all routines have a stopwatch-style *clock* associated with them, which gets reset at the beginning of that repeat of the routine. So if you have a *Routine* called trial, there will be a *clock* called trialClock and so you can get the time (in sec) from the beginning of the trial by using:

```
currentT = trialClock.getTime()
```

To see what other variables you might want to use, and also what terms you need to avoid in your chunks of code, *compile your script* before inserting the code object and take a look at the contents of that script.

Note that this page is concerned with *Code Components* specifically, and not all cases in which you might use python syntax within the Builder. It is also possible to put code into a non-code input field (such as the duration or text of a *Text Component*). The syntax there is slightly different (requiring a \$ to trigger the special handling, or \ \$ to avoid triggering special handling). The syntax to use within a *Code Component* is always regular python syntax.

Parameters

Code type: What type of code will you write?

- *Py* - Python code only (for local use)
- *JS* - Javascript only (for online use)
- *Auto* -> *JS* - Write in python code on the left and this will be auto translated to Javascript on the right.
- *Both* - write both Python and Javascript, but independently of one another (Python will be executed when you run the task locally, JS will be executed when you run the task online)

Within a *Code Component* you can write code to be executed at 6 different points within the experiment. You can use as many or as few of these as you need for any *Code Component*:

Before Experiment: Things that need to be done just once, like importing a supporting module, which do not need the experiment window to exist yet.

Begin Experiment: Things that need to be done just once, like initialising a variable for later use, which may need to refer to the experiment window.

Begin Routine: Certain things might need to be done at the start of a *Routine* e.g. at the beginning of each trial you might decide which side a stimulus will appear.

Each Frame: Things that need to be updated constantly, throughout the experiment. Note that these will be executed exactly once per video frame (on the order of every 10ms), to give dynamic displays. Static displays do not need to be updated every frame.

End Routine: At the end of the *Routine* (e.g. the trial) you may need to do additional things, like checking if the participant got the right answer

End Experiment: Use this for things like saving data to disk, presenting a graph(?), or resetting hardware to its original state.

Example code uses

1. Set a random location for your target stimulus

There are many ways to do this, but you could add the following to the *Begin Routine* section of a *Code Component* at the top of your *Routine*. Then set your stimulus position to be $(targetX, 0)$ and set the correct answer field of a *Keyboard Component* to be $corrAns$ (set both of these to update on every repeat of the Routine):

```
if random()>0.5:
    targetX=-0.5 #on the left
    corrAns='left'
else:
    targetX=0.5#on the right
    corrAns='right'
```

2. Create a patch of noise

As with the above there are many different ways to create noise, but a simple method would be to add the following to the *Begin Routine* section of a *Code Component* at the top of your *Routine*. Then set the image as $noiseTexture$:

```
noiseTexture = random.rand((128,128)) * 2.0 - 1
```

Note: Don't expect all code components to work online. Remember that code components using specific python libraries such as numpy won't smoothly translate. You might want to view the [PsychoPy to Javascript crib sheet](#) for useful info on using code components for online experiments.

3. Send a feedback message at the end of the experiment

Make a new routine, and place it at the end of the flow (i.e., the end of the experiment). Create a *Code Component* with this in the *Begin Experiment* field:

```
expClock = core.Clock()
```

and put this in the *Begin routine* field:

```
msg = "Thanks for participating - that took" + str(expClock.getTime()/60.0) +
↳ 'minutes in total'
```

Next, add a *Text Component* to the routine, and set the text to msg . Be sure that the text field's updating is set to "Set every repeat" (and not "Constant").

4. End a loop early.

Code components can also be used to control the end of a loop. For example imagine you want to end when a key response has been made 5 times:

```
if key_resp.keys: # if a key response has been made
    if len(key_resp.keys) ==5: # if 5 key presses have been made
        continueRoutine = False # end the current routine
        trials.finished = True # exit the current loop (if your loop is called "trials
↪"
```

What variables are available to use?

The most complete way to find this out for your particular script is to *compile it* and take a look at what's in there. Below are some options that appear in nearly all scripts. Remember that those variables are Python objects and can have attributes of their own. You can find out about those attributes using:

```
dir(myObject)
```

Common variables:

- `expInfo`: This is a Python Dictionary containing the information from the starting dialog box. e.g. That generally includes the 'participant' identifier. You can access that in your experiment using `exp['participant']`
- `t`: the current time (in seconds) measured from the start of this Routine
- `frameN`: the number of /completed/ frames since the start of the Routine (=0 in the first frame)
- `win`: the *Window* that the experiment is using

Your own variables:

- anything you've created in a Code Component is available for the rest of the script. (Sometimes you might need to define it at the beginning of the experiment, so that it will be available throughout.)
- the name of any other stimulus or the parameters from your file also exist as variables.
- most Components have a *status* attribute, which is useful to determine whether a stimulus has *NOT_STARTED*, *STARTED* or *FINISHED*. For example, to play a tone at the end of a Movie Component (of unknown duration) you could set start of your tone to have the 'condition'

```
myMovieName.status==FINISHED
```

Selected contents of the `numpy` library and `numpy.random` are imported by default. The entire `numpy` library is imported as `np`, so you can use a several hundred maths functions by prepending things with 'np':

- `random()`, `randint()`, `normal()`, `shuffle()` options for creating arrays of random numbers.
- `sin()`, `cos()`, `tan()`, and `pi`: For geometry and trig. By default angles are in radians, if you want the cosine of an angle specified in degrees use `cos(angle*180/pi)`, or use `numpy`'s conversion functions, `rad2deg(angle)` and `deg2rad(angle)`.
- `linspace()`: Create an array of linearly spaced values.
- `log()`, `log10()`: The natural and base-10 log functions, respectively. (It is a lowercase-L in log).
- `sum()`, `len()`: For the sum and length of a list or array. To find an average, it is better to use `average()` (due to the potential for integer division issues with `sum()/len()`).

- *average()*, *sqrt()*, *std()*: For average (mean), square root, and standard deviation, respectively. **Note:** Be sure that the numpy standard deviation formula is the one you want!
- `np._____`: Many math-related features are available through the complete numpy libraries, which are available within psychopy builder scripts as 'np.'. For example, you could use `np.hanning(3)` or `np.random.poisson(10, 10)` in a code component.

Dots (RDK) Component

The Dots Component allows you to present a Random Dot Kinematogram (RDK) to the participant of your study. Note that this component is **not yet supported for online use** (see [status of online options](#)) but users have contributed [work arounds for use online](#). These are fields of dots that drift in different directions and subjects are typically required to identify the 'global motion' of the field.

There are many ways to define the motion of the signal and noise dots. In the way the dots are configured follows [Scase, Braddick & Raymond \(1996\)](#). Although Scase et al (1996) show that the choice of algorithm for your dots actually makes relatively little difference there are some **potential** gotchas. Think carefully about whether each of these will affect your particular case:

- **limited dot lifetimes:** as your dots drift in one direction they go off the edge of the stimulus and are replaced randomly in the stimulus field. This could lead to a higher density of dots in the direction of motion providing subjects with an alternative cue to direction. Keeping dot lives relatively short prevents this.
- **noiseDots='direction':** some groups have used noise dots that appear in a random location on each frame (noiseDots='location'). This has the disadvantage that the noise dots not only have a random direction but also a random speed (whereas signal dots have a constant speed and constant direction)
- **signalDots='same':** on each frame the dots constituting the signal could be the same as on the previous frame or different. If 'different', participants could follow a single dot for a long time and calculate its average direction of motion to get the 'global' direction, because the dots would sometimes take a random direction and sometimes take the signal direction.

As a result of these, the defaults for are to have signalDots that are from a 'different' population, noise dots that have random 'direction' and a dot life of 3 frames.

Parameters

name : Everything in a experiment needs a unique name. The name should contain only letters, numbers and under-scores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See [Defining the onset/duration of components](#) for details.

stop : Governs the duration for which the stimulus is presented. See [Defining the onset/duration of components](#) for details.

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

Dot size: Size of the dots in pixel units.

fieldSize [a single value, specifying the diameter of the field (in the specified Spatial Units).] Sizes can be negative and can extend beyond the window.

fieldPos [(x,y) or [x,y]] Specifying the location of the centre of the stimulus.

spatial units [**None**, 'norm', 'cm', 'deg' or 'pix'] If None then the current units of the *Window* will be used. See *Units for the window and stimuli* for explanation of other options.

fieldShape : Defines the shape of the field in which the dots appear. For a circular field the nDots represents the *average* number of dots per frame, but on each frame this may vary a little.

Appearance

How should the stimulus look? Colour, borders, etc.

dot color : See *Color spaces*

dot color space [rgb, dkl or lms] See *Color spaces*

opacity : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

Dots

Parameters unique to the Dots component

number of dots [int] Number of dots to be generated

direction: Direction of motion for the signal dots (degrees).

speed [float] Speed of the dots (in *units* per frame)

coherence [float] Fraction moving in the signal direction on any one frame

dot life-time [int] Number of frames each dot lives for (-1=infinite)

signalDots : If 'same' then the signal and noise dots are constant. If different then the choice of which is signal and which is noise gets randomised on each frame. This corresponds to Scase et al's (1996) categories of RDK.

dot refresh rule [repeat, none] When should the sample of dots be refreshed?

noiseDots ['direction', 'position' or 'walk'] Determines the behaviour of the noise dots, taken directly from Scase et al's (1996) categories. For 'position', noise dots take a random position every frame. For 'direction' noise dots follow a random, but constant direction. For 'walk' noise dots vary their direction every frame, but keep a constant speed.

See also:

API reference for *DotStim*

Emotiv Marking Component

The Emotiv Marking component causes PsychoPy to send a marker to the EEG datastream at the time that the stimuli are presented.

For the Emotiv Marking component to work an emotiv_recording component should have already been added to the experiment.

By default markers with labels and values can be added. A time interval can be specified by sending a stop marker. If the Marker intervals overlap it is important that the labels are unique. Additionally the length of the interval must be greater than 0.2 seconds. If you need higher speeds than this, it is best to record the times of your markers manually and compare them to the times in the raw EEG data.

If you are exporting the experiment to HTML the emotiv components will have no effect in Pavlovia. To import the experiment into Emotiv OMNI, export the experiment to HTML and follow the instructions in the OMNI platform.

Parameters

Name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time to send the marker to the EEG datastream

Stop Marker: If selected the stop marker will be sent as specified by the Stop parameter. If no stop marker is sent then the marker will be an “instance” marker and will indicate a point in time. If a stop marker is sent the marker will be an “interval” marker and have a startDatetime and endDatetime associated with it.

Stop : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

marker label [string] The label assigned to this marker

marker value [int] The value assigned to this marker

stop marker [bool] Whether or not this is a stop marker. Note: stop markers were designed for relatively long intervals (of the order of one second). If you wish to mark short intervals it is safer to send two instance markers and label them appropriately so that you can create the intervals in post processing.

Emotiv Record Component

The emotiv_record component causes PsychoPy to connect to the headset so that markers can be sent to the datastream.

The emotiv_record component should be added ONCE before any stimuli have been presented at the top of first trial of the experiment.

We recommend that you use the EmotivLauncher and or EmotivPro software to establish that the headset is connected and the quality of the signals are good before running the experiment with PsychoPy.

We recommend viewing the eeg data in EmotivPro from which it can be exported as a csv or edf file. However, if you do want PsychoPy to record the data into a gzipped csv file you need to set an environment variable CORTEX_DATA=1. Additionally you will need to apply for a RAW EEG API license. See: <https://emotiv.gitbook.io/cortex-api/#prerequisites> for more details.

If you are exporting the experiment to HTML the emotiv components will have no effect in Pavlovia. To import the experiment into Emotiv OMNI, export the experiment to HTML and follow the instructions in the OMNI platform.

Getting Started

Before you can connect PsychoPy to Emotiv hardware, you need to register your AppId on the Emotiv website (<https://emotiv.com>).

Note: Normally you should **NOT** click the checkbox: “My App requires EEG access”. Otherwise you will need to apply for a RAW EEG API license.

Login to your account at emotiv.com, Goto My Account > Cortex Apps. There you will get a client_id and a client_secret that you need to copy into a file called .emotiv_creds in your home directory. One line should have “client_id” (without the quotes) then a space and then the client_id, another line should have “client_secret” (without the quotes and then a space and then the client secret. A line beginning with a hash will be ignored. eg

```
—begin file —  
# client_id and client_secret for Emotiv application  
client_id abcd1234...
```

```
client_secret wxyz78910...
—end file—
```

Troubleshooting

- Check that the .emotiv_creds file does not have “.txt” file extension.
- Ensure the file format is exactly correct (do **not** include the begin and end file lines)
- Ensure that your AppId does not require EEG data **or** apply for RAW EEG API access through EMOTIV support.
- Ensure you connect your headset using EmotivPro or EmotivLauncher before you run the experiment.

Parameters

Name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : Set this to 0

Stop : Set this to 1 seconds

Setting these values just allows the routine to finish

Eye Tracker Region of Interest Component

Please note: This is a new component, and is subject to change.

Record eye movement events occurring within a defined Region of Interest (ROI). Note that you will still need to add an Eyetracker Record component to this routine to save eye movement data.

Parameters

Basic

Name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start [float or integer] The time that the stimulus should first play. See *Defining the onset/duration of components* for details.

Stop (duration): The length of time (sec) to record for. An *expected duration* can be given for visualisation purposes. See *Defining the onset/duration of components* for details; note that only seconds are allowed.

Shape: A shape to outline the Region of Interest. Same as the *Polygon (shape) Component*. Using a regular polygon allows you to specify the number of vertices (a circle would be a regular polygon with a large number of vertices e.g. 100). Using *custom polygon* allows you to add a list of coordinates to build custom shapes.

End Routine On: What event, if any, do you want to end the current routine. If “look at” or “look away” selected you should also specify the minimum look time in milliseconds that will constitute an event of interest.

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

ori [degrees] The orientation of the entire patch (texture and mask) in degrees.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

size [(width, height)] Size of the stimulus on screen

spatial units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

Data

Save onset/offset times: bool Whether to save the onset and offset times of the component.

Save...: What eye movement events do you want to save? *Every Look* will return a list of looks; *First Look* and *Last Look* will return the first and last looks respectively.

Time Relative To: What do you want the timing of the timestamped events to be relative to?

See also:

API reference for ROI

Eye Tracker Calibration Component (Standalone Routine)

Please note: This is a new component, and is subject to change.

Note that the Eye tracking calibration component is a “standalone routine”, this means that rather than generating a component that is added to an existing routine, it is a routine in itself, that is then placed along your flow. The reason for this implementation is that calibration represents a series of events that will be relatively uniform across studies, and often we would not want to add any additional info to this phase of the study (i.e. images, text etc.)

Parameters

Basic

Name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Target Layout: How many targets do you want to be presented for calibration? Points will be displayed in a grid.

Randomise Target Positions: bool If `True` the point positions will be presented in a random order.

Target

Aesthetic features of the calibration target.

Outer Fill Color [string] The color of the outer circle of the target. None/Blank will be transparent.

Outer Border Color [string] The color of the border of the outer circle of the target.

Inner Fill Color [string] The color of the inner circle of the target. None/Blank will be transparent.

Inner Border Color [string] The color of the border of the inner circle of the target.

Color Space : The color space in which to read the defined colors.

Outer Border Width [int] The width of the line around the outer target.

Animation

How should the animation of the calibration routine appear?

Progress Mode : Should each target appear one after the other and progress based on time? Or should the next target be presented once the space key has been pressed.

Target Duration [int or float] The duration of the pulse of the outer circle (i.e. time or expand + contract)

Expand Scale: How much larger should the outer circle get?

Animate Position Changes: **bool** Should the target appear as though it is moving across the screen from one location to the next?

Movement Duration: **int or float** The duration of the movement from one point to the next.

See also:

API reference for `EyetrackerCalibration`

Eye Tracker Record Component

Please note: This is a new component, and is subject to change.

The eye-tracker record component provides a way to record eye movement data within an experiment. To do so, specify the starting time relative to the start of the routine (see *start* below) and a stop time (= duration in seconds). Before using the eye-tracking record component, you must specify your eye tracking device under *experiment settings* > *Eyetracking*. Here the available options are:

- GazePoint
- MouseGaze
- SR Research Ltd (aka EyeLink)
- Tobii Technology

If you are developing your eye-tracking paradigm out-of-lab we recommend using *MouseGaze* which will simulate eye movement responses through monitoring your mouse cursor and buttons to simulate movements and blinks.

The resulting eye-movement coordinates are stored and accessible through calling *etRecord.pos* where *etRecord* corresponds to the name of the eye-tracking record component, you can set something (e.g. a polygon) to be in the same location as the current “look” by setting the position field to `:code: `etRecord.pos` and setting the field to update on **every frame** When running an eye tracking study, you can optionally save the data in hdf5 format through selecting this option in the experiment settings > data tab.

Parameters

Basic

name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the stimulus should first play. See *Defining the onset/duration of components* for details.

stop (duration): The length of time (sec) to record for. An *expected duration* can be given for visualisation purposes. See *Defining the onset/duration of components* for details; note that only seconds are allowed.

Data

Save onset/offset times: bool Whether to save the onset and offset times of the component.

Sync timing with screen refresh: bool Whether to sync the start time of the component with the window refresh.

See also:

API reference for *EyeTracker*

Eye Tracker Validation Component (Standalone Routine)

Please note: This is a new component, and is subject to change.

The Eye tracking validation component is also a “standalone routine”, this means that rather than generating a component that is added to an existing routine, it is a routine in itself, that is then placed along your flow. The reason for this implementation is that calibration/validation represent a series of events that will be relatively uniform across studies, and often we would not want to add any additional info to this phase of the study (i.e. images, text etc.)

Parameters

Basic

Name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Target Layout: How many targets do you want to be presented for calibration? Points will be displayed in a grid.

Randomise Target Positions: bool If `True` the point positions will be presented in a random order.

Gaze Cursor Color: The color of the gaze cursor.

Target

Aesthetic features of the target.

Outer Fill Color [string] The color of the outer circle of the target. `None/Blank` will be transparent.

Outer Border Color [string] The color of the border of the outer circle of the target.

Inner Fill Color [string] The color of the inner circle of the target. `None/Blank` will be transparent.

Inner Border Color [string] The color of the border of the inner circle of the target.

Color Space : The color space in which to read the defined colors.

Outer Border Width [int] The width of the line around the outer target.

Animation

How should the animation of the validation routine appear?

Progress Mode : Should each target appear one after the other and progress based on time? Or should the next target be presented once the space key has been pressed.

Target Duration: int or float The duration of the pulse of the outer circle (i.e. time or expand + contract)

Expand Scale: How much larger should the outer circle get?

Animate Position Changes: bool Should the target appear as though it is moving across the screen from one location to the next?

Movement Duration: int or float The duration of the movement from one point to the next.

Data

Save As Image Save the results as an image

Show Results Screen Show a screen with the results after completion

See also:

API reference for `EyetrackerCalibration`

Form Component

Please note that this component is still in Beta mode and is therefore developing

The Form component enables Psychopy to be used as a questionnaire tool, where participants can be presented with a series of questions requiring responses. Form items, defined as questions and response pairs, are presented simultaneously onscreen with a scrollable viewing window.

Properties

Name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start [int, float] The time that the stimulus should first appear.

Stop [int, float] Governs the duration for which the stimulus is presented.

Items [A csv / xlsx file **To get started, we recommend selecting the “Open/Create Icon” which will open up a template forms spreadsheet** A csv/xlsx file should have the following key, value pairs / column headers:]

index The item index as a number

itemText The item question string

itemWidth The question width between 0 : 1

type The type of rating e.g., ‘choice’, ‘rating’, ‘slider’, ‘free-text’

responseWidth The question width between 0 : 1

options A sequence of tick labels for options e.g., yes, no

layout Response object layout e.g., ‘horiz’ or ‘vert’

itemColor The question text font color

responseColor The response object color

granularity If you are using a slider, what do you want the granularity of the slider to be?

Missing column headers will be replaced by default entries, with the exception of *itemText* and *type*, which are required. The def

index 0 (increments for each item)

itemWidth 0.7

responseWidth 0.3

options Yes, No

layout horiz

itemColor from style

responseColor from style

Data format [menu] Choose whether to store items data by column or row in your datafile.

randomize [bool] Randomize order of Form elements

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

Size [[X,Y]] Size of the stimulus, to be specified in 'height' units.

Pos [[X,Y]] The position of the centre of the stimulus, to be specified in 'height' units.

Item padding [float] Space or padding between Form elements (i.e., question and response text), to be specified in 'height' units.

Appearance

How should the stimulus look? Color, borders, etc. Many of these read-only parameters become editable when *Styles* is set to *custom*.

style [light, dark] Whether to style items in your form for a light or a dark background

border color [color] See *Color spaces*

fill color [color] See *Color spaces*

opacity : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

Formatting

Formatting text

Text height [float] Text height of the Form elements (i.e., question and response text).

Font Font to use in text.

Note: Top tip: Form has an attribute to check if all questions have been answered `form.complete`. You could use this to make a "submit" button appear only when the form is completed!

See also:

API reference for *Form*

Creating a Google Cloud Speech API key

There are a few steps but they’re relatively easy. Pricing is free for the first 60 minutes per month and 1-2cents per minute after that Information here: <https://cloud.google.com/speech-to-text>

Note that You might be asked to enter card details but you are not charged an auto update unless you manually enter the card details when prompted

Steps

- Create an account on [Google Cloud Platform](#) (this is not the same as simply gmail or Google Worksuite)
- Create a project from here: <https://console.cloud.google.com/home/dashboard> by selecting manage resources > create project The projects could just be for the entire lab, say, or for each experiment, depending on the granularity you need for billing (We believe)
- Enable the Speech API for that project: select the project in the manage resources page, go to <https://console.cloud.google.com/apis/library/speech.googleapis.com> click “enable”.
- Then click on Credentials and create Service Account credentials.

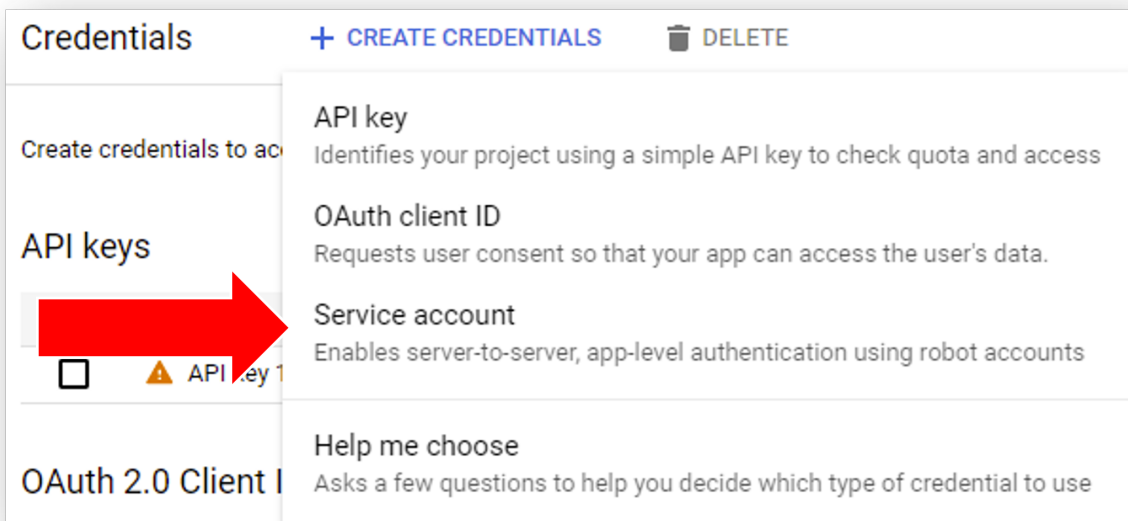


Fig. 5.1: Add credentials to your Google cloud project and select “Service Account”.

- Grant the service account access to Google Speech Client.
- Once you have your service account set up you can add a key and make a downloadable JSON file. Store it somewhere (private) on your computer. You don’t need to go through these steps for every new project - once you have a key you can use it for all of your projects.

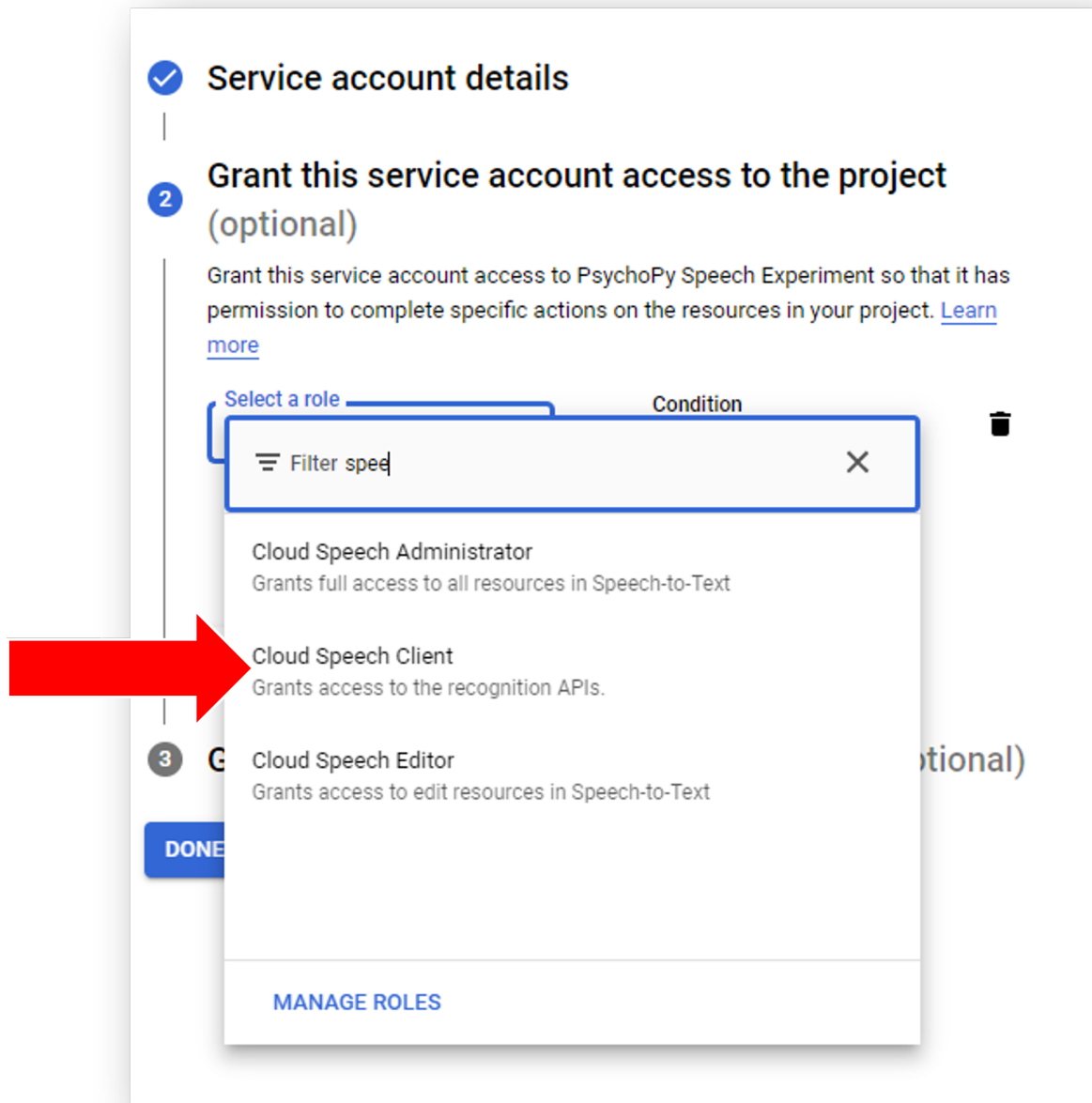


Fig. 5.2: Search for “Google Speech Client” and give this account access to that API.

Warning: Be careful not to store the json file in the same location as any experiment folder that might later be shared on - this is a private file - so keep it somewhere safe.

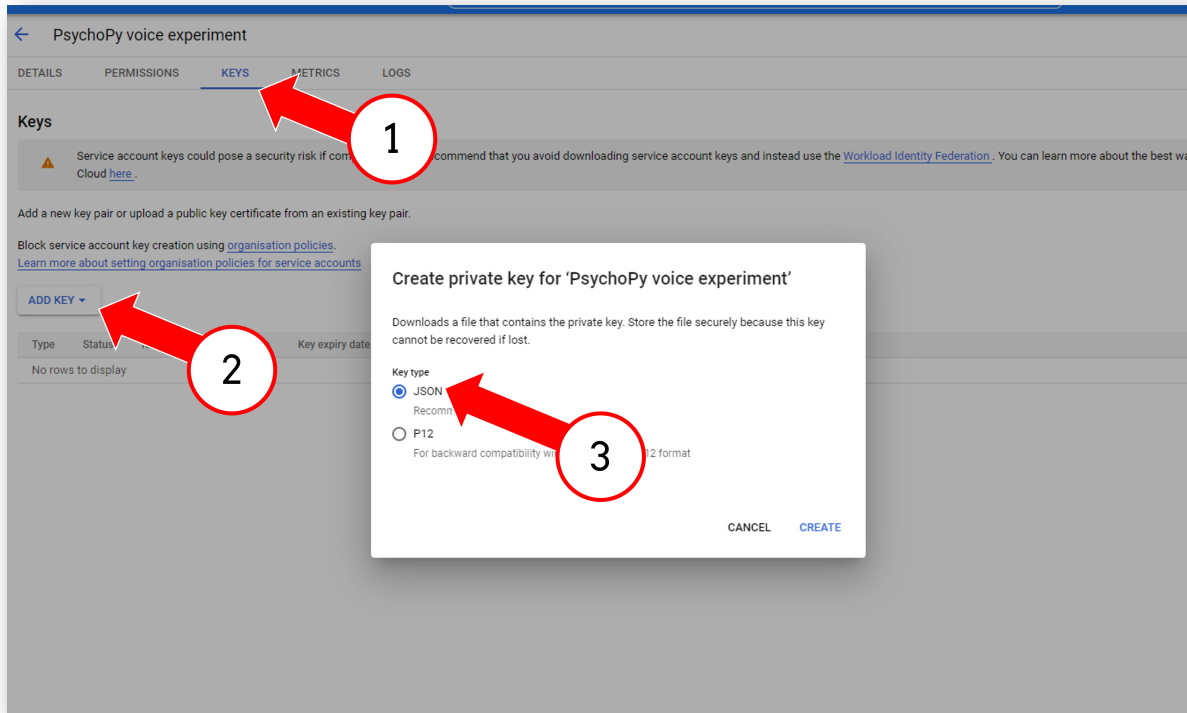


Fig. 5.3: Generate a downloadable JSON for this project.

- Finally, in go fo File > Preferences and add the path to the JSON file in General > appKeyGoogleCloud.

Warning: Remember to check that your accounts billing information stays up to date. Even if you haven't done enough recordings to warrant a large payment, if a card on your billing account expires this will invalidate the JSON key and raise a "billing" error in .

Grating Component

The Grating stimulus allows a texture to be wrapped/cycled in 2 dimensions, optionally in conjunction with a mask (e.g. Gaussian window). The texture can be a bitmap image from a variety of standard file formats, or a synthetic texture such as a sinusoidal grating. The mask can also be derived from either an image, or mathematical form such as a Gaussian.

When using gratings, if you want to use the *spatial frequency* setting then create just a single cycle of your texture and allow to handle the repetition of that texture (do not create the cycles you're expecting within the texture).

Gratings can have their position, orientation, size and other settings manipulated on a frame-by-frame basis. There is a performance advantage (in terms of milliseconds) to using images which are square and powers of two (32, 64, 128, etc.), however this is slight and would not be noticed in the majority of experiments.

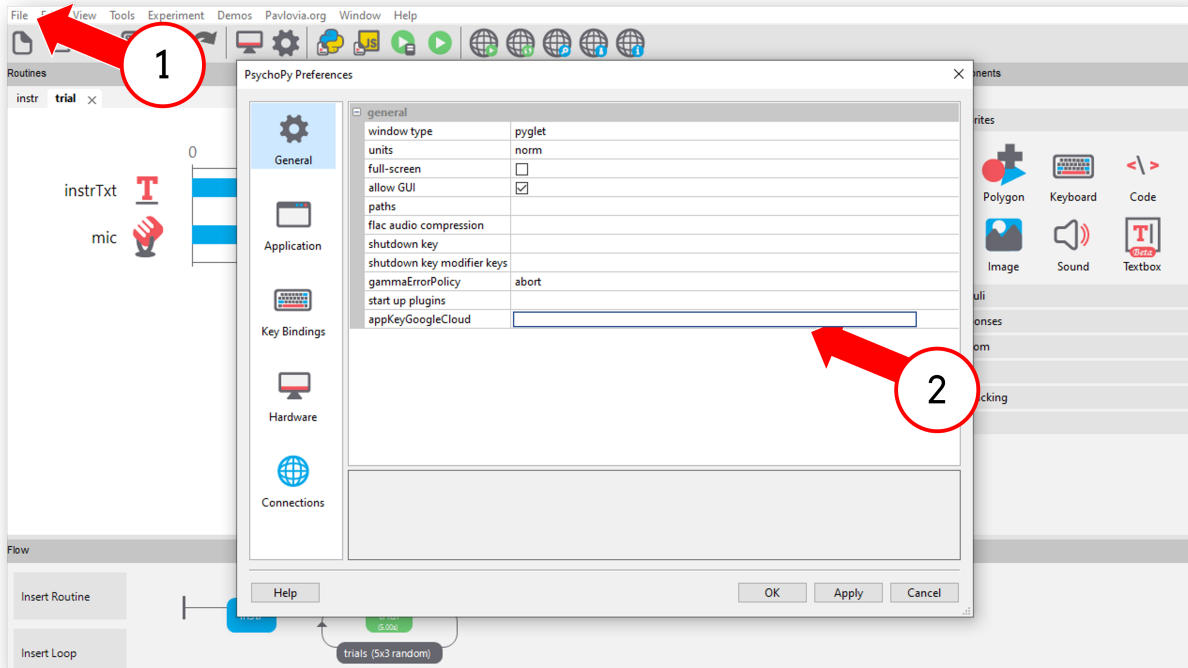


Fig. 5.4: Setup your preferences to use your downloaded JSON - this will apply to all experiments using the mic - not just this experiment.

Parameters

Name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

Stop : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

Appearance

How should the stimulus look? Colour, borders, etc.

blend mode [average, add] How should colours blend when overlaid onto something? Should colours be averaged, or added?

foreground color : See *Color spaces*

foreground color space [rgb, dkl or lms] See *Color spaces*

Opacity [0-1] Can be used to create semi-transparent gratings

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

Orientation [degrees] The orientation of the entire patch (texture and mask) in degrees.

Position [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

Size [[size_x, size_y] or a single value (applied to x and y)] The size of the stimulus in the given units of the stimulus/window. If the mask is a Gaussian then the size refers to width at 3 standard deviations on either side of the mean (i.e. $sd=size/6$)

Units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

Texture

Control how the stimulus handles textures.

Texture: a filename, a standard name (*sin*, *sqr*) or a variable giving a numpy array This specifies the image that will be used as the *texture* for the visual patch. The image can be repeated on the patch (in either x or y or both) by setting the spatial frequency to be high (or can be stretched so that only a subset of the image appears by setting the spatial frequency to be low). Filenames can be relative or absolute paths and can refer to most image formats (e.g. tif, jpg, bmp, png, etc.). If this is set to none, the patch will be a flat colour.

Mask [a filename, a standard name (*gauss*, *circle*, *raisedCos*) or a numpy array of dimensions NxNx1] The mask can define the shape (e.g. *circle* will make the patch circular) or something which overlays the patch e.g. noise.

Interpolate : If *linear* is selected then linear interpolation will be applied when the image is rescaled to the appropriate size for the screen. *Nearest* will use a nearest-neighbour rule.

Phase [single float or pair of values [X,Y]] The position of the texture within the mask, in both X and Y. If a single value is given it will be applied to both dimensions. The phase has units of cycles (rather than degrees or radians), wrapping at 1. As a result, setting the phase to 0,1,2... is equivalent, causing the texture to be centered on the mask. A phase of 0.25 will cause the image to shift by half a cycle (equivalent to pi radians). The advantage of this is that if you set the phase according to time it is automatically in Hz.

Spatial Frequency [[SF_x, SF_y] or a single value (applied to x and y)] The spatial frequency of the texture on the patch. The units are dependent on the specified units for the stimulus/window; if the units are *deg* then the SF units will be *cycles/deg*, if units are *norm* then the SF units will be cycles per stimulus. If this is set to none then only one cycle will be displayed.

Texture Resolution [an integer (power of two)] Defines the size of the resolution of the texture for standard textures such as *sin*, *sqr* etc. For most cases a value of 256 pixels will suffice, but if stimuli are going to be very small then a lower resolution will use less memory.

See also:

API reference for *GratingStim*

Image Component

The Image stimulus allows an image to be presented, which can be a bitmap image from a variety of standard file formats, with an optional transparency mask that can effectively control the shape of the image. The mask can also be derived from an image file, or mathematical form such as a Gaussian.

It is a really good idea to get your image in roughly the size (in pixels) that it will appear on screen to save memory. If you leave the resolution at 12 megapixel camera, as taken from your camera, but then present it on a standard screen at 1680x1050 (=1.6 megapixels) then PsychoPy and your graphics card have to do an awful lot of unnecessary work. There is a performance advantage (in terms of milliseconds) to using images which are square and powers of two (32, 64, 128, etc.), but this is slight and would not be noticed in the majority of experiments.

Images can have their position, orientation, size and other settings manipulated on a frame-by-frame basis.

Parameters

Name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

Stop : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

Image [a filename or a standard name (sin, sqr)] Filenames can be relative or absolute paths and can refer to most image formats (e.g. tif, jpg, bmp, png, etc.). If this is set to none, the patch will be a flat colour.

Appearance

How should the stimulus look? Colour, borders, etc.

opacity [value from 0 to 1] If opacity is reduced then the underlying images/stimuli will show through

foreground color [Colors can be applied to luminance-only images (not to rgb images)] See *Color spaces*

foreground color space [to be used if a color is supplied] See *Color spaces*

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

Position [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

Size [[sizex, sizey] or a single value (applied to x and y)] The size of the stimulus in the given units of the stimulus/window. If the mask is a Gaussian then the size refers to width at 3 standard deviations on either side of the mean (i.e. $sd=size/6$) Set this to be blank to get the image in its native size.

Orientation [degrees] The orientation of the entire patch (texture and mask) in degrees.

Units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

flip horizontally [bool] Flip the image along the horizontal axis

flip vertically [bool] Flip the image along the vertical axis

spatial units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

Texture

Control how the stimulus handles textures.

Mask [a filename, a standard name (gauss, circle, raisedCos) or a numpy array of dimensions NxNx1] The mask can define the shape (e.g. circle will make the patch circular) or something which overlays the patch e.g. noise.

Interpolate : If *linear* is selected then linear interpolation will be applied when the image is rescaled to the appropriate size for the screen. *Nearest* will use a nearest-neighbour rule.

Texture Resolution: This is only needed if you use a synthetic texture (e.g. sinusoidal grating) as the image.

See also:

API reference for *ImageStim*

ioLab Systems buttonbox Component

A button box is a hardware device that is used to collect participant responses with high temporal precision, ideally with true ms accuracy.

Both the response (which button was pressed) and time taken to make it are returned. The time taken is determined by a clock on the device itself. This is what makes it capable (in theory) of high precision timing.

Check the log file to see how long it takes for to reset the button box's internal clock. If this takes a while, then the RT timing values are not likely to be high precision. It might be possible for you to obtain a correction factor for your computer + button box set up, if the timing delay is highly reliable.

The ioLabs button box also has a built-in voice-key, but does not have an interface for it. Use a microphone component instead.

Properties

name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : The duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

Force end of Routine [checkbox] If this is checked, the first response will end the routine.

Data

What information to save, how to lay it out and when to save it.

Active buttons [None, or an integer, list, or tuple of integers 0-7] The ioLabs box lets you specify a set of active buttons. Responses on non-active buttons are ignored by the box, and never sent to . This field lets you specify which buttons (None, or some or all of 0 through 7).

Store [(choice of: first, last, all, nothing)] Which button events to save in the data file. Events and the response times are saved, with RT being recorded by the button box (not by).

Store correct [checkbox] If selected, a correctness value will be saved in the data file, based on a match with the given correct answer.

Discard previous [checkbox] If selected, any previous responses will be ignored (typically this is what you want).

Hardware

Parameters for controlling hardware.

Lights off [checkbox] If selected, all lights will be turned off at the end of each routine.

Lights : If selected, the lights above the active buttons will be turned on.

Using code components, it is possible to turn on and off specific lights within a trial. See the API for `iolab`.

See also:

API reference for `iolab`

JoyButtons Component

The JoyButtons component can be used to collect gamepad/joystick button responses from a participant.

By not storing the button number pressed and checking the *forceEndTrial* box it can be used simply to end a *Routine*. If no gamepad/joystick is installed the keyboard can be used to simulate button presses by pressing 'ctrl' + 'alt' + digit(0-9).

Parameters

Name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start [float or integer] The time that joyButtons should first get checked. See *Defining the onset/duration of components* for details.

Stop [float or integer] When joyButtons should no longer get checked. See *Defining the onset/duration of components* for details.

Force end routine : If this box is checked then the *Routine* will end as soon as one of the *allowed* buttons is pressed.

Data

What information to save, how to lay it out and when to save it.

Allowed buttons : A list of allowed buttons can be specified here, e.g. [0,1,2,3], or the name of a variable holding such a list. If this box is left blank then any button that is pressed will be read. Only *allowed buttons* count as having been pressed; any other button will not be stored and will not force the end of the Routine. Note that button numbers (0, 1, 2, 3, ...), should be separated by commas.

Store : Which button press, if any, should be stored; the first to be pressed, the last to be pressed or all that have been pressed. If the button press is to force the end of the trial then this setting is unlikely to be necessary, unless two buttons happen to be pressed in the same video frame. The response time will also be stored if a button press is recorded. This time will be taken from the start of joyButtons checking (e.g. if the joyButtons was initiated 2 seconds into the trial and a button was pressed 3.2s into the trials the response time will be recorded as 1.2s).

Store correct : Check this box if you wish to store whether or not this button press was correct. If so then fill in the next box that defines what would constitute a correct answer e.g. 1 or *ScorrAns* (note this should not be in inverted commas). This is given as Python code that should return True (1) or False (0). Often this correct answer will be defined in the settings of the *Loops*.

Hardware

Parameters for controlling hardware.

Device number [integer] Which gamepad/joystick device number to use. The first device found is numbered 0.

Joystick Component

The Joystick component can be used to collect responses from a participant. The coordinates of the joystick location are given in the same coordinates as the Window, with (0,0) in the centre. Coordinates are correctly scaled for 'norm' and 'height' units. User defined scaling can be set by updating joystick.xFactor and joystick.yFactor to the desired values. Joystick.device.getX() and joystick.device.getY() always return 'norm' units. Joystick.getX() and joystick.getY() are scaled by xFactor or yFactor

No cursor is drawn to represent the joystick current position, but this is easily provided by updating the position of a partially transparent '.png' image on each screen frame using the joystick coordinates: joystick.getX() and joystick.getY(). To ensure that the cursor image is drawn on top of other images it should be the last image in the trial.

Joystick Emulation If no joystick device is found, the mouse and keyboard are used to emulate a joystick device. Joystick position corresponds to mouse position and mouse buttons correspond to joystick buttons (0,1,2). Other buttons can be simulated with key chords: 'ctrl' + 'alt' + digit(0..9).

Scenarios

This can be used in various ways. Here are some scenarios (email the list if you have other uses for your joystick):

Use the joystick to record the location of a button press

Use the joystick to control stimulus parameters Imagine you want to use your joystick to make your 'patch' _ bigger or smaller and save the final size. Call your *joystickComponent* 'joystick', set it to save its state at the end of the trial and set the button press to end the Routine. Then for the size setting of your Patch stimulus insert *\$joystick.getX()* to use the x position of the joystick to control the size or *\$joystick.getY()* to use the y position.

Tracking the entire path of the joystick during a period

Parameters Basic

Name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the joystick should first be checked. See *Defining the onset/duration of components* for details.

stop : When the joystick is no longer checked. See *Defining the onset/duration of components* for details.

Force End Routine on Press If this box is checked then the *Routine* will end as soon as one of the joystick buttons is pressed.

Data

What information to save, how to lay it out and when to save it.

Save Joystick State How often do you need to save the state of the joystick? Every time the subject presses a joystick button, at the end of the trial, or every single frame? Note that the text output for cases where you store the joystick data repeatedly per trial (e.g. every press or every frame) is likely to be very hard to interpret, so you may then need to analyse your data using the psydat file (with python code) instead. Hopefully in future releases the output of the text file will be improved.

Time Relative To Whenever the joystick state is saved (e.g. on button press or at end of trial) a time is saved too. Do you want this time to be relative to start of the *Routine*, or the start of the whole experiment?

Clickable Stimulus A comma-separated list of your stimulus names that ‘can be “clicked” by the participant. e.g. target, foil.

Store params for clicked The params (e.g. name, text), for which you want to store the current value, for the stimulus that was “clicked” by the joystick. Make sure that all the clickable objects have all these params.

Allowed Buttons Joystick buttons accepted for input (blank for any) numbers separated by ‘commas’.

Hardware

Parameters for controlling hardware.

Device Number If you have multiple joystick/gamepad devices which one do you want (0, 1, 2, ...).

See also:

API reference for `Joystick`

Keyboard Component

The Keyboard component can be used to collect responses from a participant.

By not storing the key press and checking the *forceEndTrial* box it can be used simply to end a *Routine*

Parameters

name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the keyboard should first get checked. See *Defining the onset/duration of components* for details.

stop : When the keyboard is no longer checked. See *Defining the onset/duration of components* for details.

force end routine If this box is checked then the *Routine* will end as soon as one of the *allowed* keys is pressed.

Data

What information to save, how to lay it out and when to save it.

allowed keys [list] A list of allowed keys can be specified here, e.g. ['m','z','1','2'], or the name of a variable holding such a list. If this box is left blank then any key that is pressed will be read. Only *allowed keys* count as having been pressed; any other key will not be stored and will not force the end of the Routine. Note that key names (even for number keys) should be given in single quotes, separated by commas. Cursor control keys can be accessed with 'up', 'down', and so on; the space bar is 'space'. To find other special keys, run the Coder Input demo, "what_key.py", press the key, and check the Coder output window.

store [last key, first key, all keys, nothing] Which key press, if any, should be stored; the first to be pressed, the last to be pressed or all that have been pressed. If the key press is to force the end of the trial then this setting is unlikely to be necessary, unless two keys happen to be pressed in the same video frame. The response time will also be stored if a keypress is recorded. This time will be taken from the start of keyboard checking (e.g. if the keyboard was initiated 2 seconds into the trial and a key was pressed 3.2s into the trials the response time will be recorded as 1.2s).

store correct [bool] Check this box if you wish to store whether or not this key press was correct. If so then fill in the next box that defines what would constitute a correct answer e.g. left, 1 or *\$corrAns* (note this should not be in inverted commas). This is given as Python code that should return True (1) or False (0). Often this correct answer will be defined in the settings of the *Loops*.

discard previous [bool] Check this box to ensure that only key presses that occur during this keyboard checking period are used. If this box is not checked a keyboard press that has occurred before the start of the checking period will be interpreted as the first keyboard press. For most experiments this box should be checked.

See also:

API reference for *psychopy.event*

Microphone Component

Enabled for online use in version 2021.2 onwards

The microphone component provides a way to record sound during an experiment. You can even transcribe the recording to text! Take a look at the documentation on *Creating a Google Cloud Speech API key* to get started with that.

When using a mic recording, specify the starting time relative to the start of the routine (see *start* below) and a stop time (= duration in seconds). A blank duration evaluates to recording for 0.000s.

The resulting sound files are saved in .wav format (at the specified sampling frequency), one file per recording. The files appear in a new folder within the data directory (the subdirectory name ends in *_wav*). The file names include the unix (epoch) time of the onset of the recording with milliseconds, e.g., *mic-1346437545.759.wav*.

It is possible to stop a recording that is in progress by using a code component. Every frame, check for a condition (such as key 'q', or a mouse click), and call the *mic.stop()* method of the microphone component. The recording will end at that point and be saved.

Parameters

Basic

name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the stimulus should first play. See *Defining the onset/duration of components* for details.

stop (duration): The length of time (sec) to record for. An *expected duration* can be given for visualisation purposes. See *Defining the onset/duration of components* for details; note that only seconds are allowed.

Device: Which microphone device to use

Transcription

Transcribe Audio: bool Whether to transcribe audio recordings and store the data

Online Transcription Backend: What transcription service to use to transcribe audio [Google](#) or built in. Note that in our experience Google has better transcription results, though we *highly* recommend taking a look at the documentation on *Creating a Google Cloud Speech API key* to get started.

Transcription Language: string The language code for your chosen transcription language e.g. English (United States) is “en-US” see [list of codes here](#)

Expected Words: list of strings A list of key words that you want to listen for e.g. [*“Hello”, “World”*] if blank all words will be listened for.

Data

Save onset/offset times: bool Whether to save the onset and offset times of the component.

Sync timing with screen refresh: bool Whether to sync the start time of the component with the window refresh.

Output File Type: File type to save audio as (default is wav).

Speaking Start/Stop Times: bool Save onset/offset of speech.

Trim Silent: bool Trim periods of silent from the output file.

Hardware

Channels: Record 1 (mono) or 2 (stereo) channels (auto will save as many as the recording device allows).

Sample Rate (Hz): Sampling rate of recorded audio.

Max Recording Size (kb): Max recording size to avoid excessively large output files.

See also:

API reference for *AdvAudioCapture* API reference for `transcribe`

Mouse Component

The Mouse component can be used to collect responses from a participant. The coordinates of the mouse location are given in the same coordinates as the Window, with (0,0) in the centre.

Scenarios

This can be used in various ways. Here are some scenarios (email the list if you have other uses for your mouse):

Use the mouse to record the location of a button press

Use the mouse to control stimulus parameters Imagine you want to use your mouse to make your 'patch' bigger or smaller and save the final size. Call your *mouse* 'mouse', set it to save its state at the end of the trial and set the button press to end the Routine. Then for the size setting of your Patch stimulus insert `$mouse.getPos()[0]` to use the x position of the mouse to control the size or `$mouse.getPos()[1]` to use the y position.

Tracking the entire path of the mouse during a period

Parameters

name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the mouse should first be checked. See *Defining the onset/duration of components* for details.

stop : When the mouse is no longer checked. See *Defining the onset/duration of components* for details.

Force End Routine on Press If this box is checked then the *Routine* will end as soon as one of the mouse buttons is pressed.

Data

What information to save, how to lay it out and when to save it.

save mouse state How often do you need to save the state of the mouse? Every time the subject presses a mouse button, at the end of the trial, or every single frame? Note that the text output for cases where you store the mouse data repeatedly per trial (e.g. every press or every frame) is likely to be very hard to interpret, so you may then need to analyse your data using the psydat file (with python code) instead. Hopefully in future releases the output of the text file will be improved.

time relative to Whenever the mouse state is saved (e.g. on button press or at end of trial) a time is saved too. Do you want this time to be relative to start of the *Routine*, or the start of the whole experiment?

new clicks only [bool] Store only new clicks

clickable stimuli [list] List of stimulus names within the same routine which can be clicked on

store params for clicked [list] List of parameter names to store from stimuli which are clicked on

See also:

API reference for *Mouse*

Movie Component

The Movie component allows movie files to be played from a variety of formats (e.g. mpeg, avi, mov).

The movie can be positioned, rotated, flipped and stretched to any size on the screen (using the *Units for the window and stimuli* given).

Parameters

name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : Governs the duration for which the stimulus is presented (if you want to cut a movie short). Usually you can leave this blank and insert the *Expected* duration just for visualisation purposes. See *Defining the onset/duration of components* for details.

movie [string] The filename of the movie, including the path. The path can be absolute or relative to the location of the experiment (.psyexp) file.

forceEndRoutine : If checked, when the movie finishes, the routine will be ended.

Appearance

How should the stimulus look? Colour, borders, etc.

opacity [float] Vary the transparency, from 0.0 = invisible to 1.0 = opaque

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

ori [degrees] Movies can be rotated in real-time too! This specifies the orientation of the movie in degrees.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

size [[size_x, size_y] or a single value (applied to both x and y)] The size of the stimulus in the given units of the stimulus/window.

spatial units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

Playback

How should stimulus play? Speed, volume, etc.

backend [moviepy, avbin, opencv] What Python package should be used to play the movie?

no audio [bool] Tick to mute audio

loop playback [bool] Should video loop on completion?

See also:

API reference for *MovieStim*

Parallel Port Out Component

This component allows you to send triggers to a parallel port, USB2TTL8, or LabJack U3 device.

An example usage would be in EEG experiments to set the port to 0 when no stimuli are present and then set it to an identifier value for each stimulus synchronised to the start/stop of that stimulus. In that case you might set the *Start data* to be *\$ID* (with ID being a column in your conditions file) and set the *Stop Data* to be 0.

Properties

Name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

Stop : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

Data

What information to save, how to lay it out and when to save it.

Start data [0-255] When the start time/condition occurs this value will be sent to the parallel port. The value is given as a byte (a value from 0-255) controlling the 8 data pins of the parallel port.

Stop data [0-255] As with start data but sent at the end of the period.

Sync to screen [boolean] If true then the parallel port will be sent synchronised to the next screen refresh, which is ideal if it should indicate the onset of a visual stimulus. If set to False then the data will be set on the parallel port immediately.

Hardware

Parameters for controlling hardware.

Port address [select the appropriate option] You need to know the address of the parallel port you wish to write to. The options that appear in this drop-down list are determined by the application preferences. You can add your particular port there if you prefer.

Register [U3 register to write to] When using a LabJack U3, you can select which register is used to write a data byte to. Register EIO is the default.

See also:

API reference for U3

Patch (image) Component

The Patch stimulus allows images to be presented in a variety of forms on the screen. It allows the combination of an image, which can be a bitmap image from a variety of standard file formats, or a synthetic repeating texture such as a sinusoidal grating. A transparency mask can also be control the shape of the image, and this can also be derived from either a second image, or mathematical form such as a Gaussian.

Patches can have their position, orientation, size and other settings manipulated on a frame-by-frame basis. There is a performance advantage (in terms of milliseconds) to using images which are square and powers of two (32, 64, 128, etc.), however this is slight and would not be noticed in the majority of experiments.

Parameters

name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

image [a filename, a standard name ('sin', 'sqr') or a numpy array of dimensions NxNx1 or NxNx3] This specifies the image that will be used as the *texture* for the visual patch. The image can be repeated on the patch (in either x or y or both) by setting the spatial frequency to be high (or can be stretched so that only a subset of the image appears by setting the spatial frequency to be low). Filenames can be relative or absolute paths and can refer to most image formats (e.g. tif, jpg, bmp, png, etc.). If this is set to none, the patch will be a flat colour.

mask [a filename, a standard name ('gauss', 'circle') or a numpy array of dimensions NxNx1] The mask can define the shape (e.g. circle will make the patch circular) or something which overlays the patch e.g. noise.

ori [degrees] The orientation of the entire patch (texture and mask) in degrees.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

size [[size_x, size_y] or a single value (applied to x and y)] The size of the stimulus in the given units of the stimulus/window. If the mask is a Gaussian then the size refers to width at 3 standard deviations on either side of the mean (i.e. $sd=size/6$)

units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

Advanced Settings

colour : See *Color spaces*

colour space [rgb, dkl or lms] See *Color spaces*

SF [[SF_x, SF_y] or a single value (applied to x and y)] The spatial frequency of the texture on the patch. The units are dependent on the specified units for the stimulus/window; if the units are *deg* then the SF units will be *cycles/deg*, if units are *norm* then the SF units will be cycles per stimulus. If this is set to none then only one cycle will be displayed.

phase [single float or pair of values [X,Y]] The position of the texture within the mask, in both X and Y. If a single value is given it will be applied to both dimensions. The phase has units of cycles (rather than degrees or radians), wrapping at 1. As a result, setting the phase to 0,1,2... is equivalent, causing the texture to be centered on the mask. A phase of 0.25 will cause the image to shift by half a cycle (equivalent to pi radians). The advantage of this is that if you set the phase according to time it is automatically in Hz.

Texture Resolution [an integer (power of two)] Defines the size of the resolution of the texture for standard textures such as *sin*, *sqr* etc. For most cases a value of 256 pixels will suffice, but if stimuli are going to be very small then a lower resolution will use less memory.

interpolate : If *linear* is selected then linear interpolation will be applied when the image is rescaled to the appropriate size for the screen. *Nearest* will use a nearest-neighbour rule.

See also:

API reference for *PatchStim*

Polygon (shape) Component

(added in version 1.78.00)

The Polygon stimulus allows you to present a wide range of regular geometric shapes. The basic control comes from setting the

- 2 vertices give a line
- 3 give a triangle
- 4 give a rectangle etc.
- a large number will approximate a circle/ellipse

The size parameter takes two values. For a line only the first is used (then use *ori* to specify the orientation). For triangles and rectangles the size specifies the height and width as expected. Note that for pentagons upwards, however, the size determines the width/height of the ellipse on which the vertices will fall, rather than the width/height of the vertices themselves (slightly smaller typically).

Parameters

name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

shape [line, triangle, rectangle, cross, star, regular polygon] What shape the stimulus is

num vertices [int] The number of vertices for your shape (2 gives a line, 3 gives a triangle, . . . a large number results in a circle/ellipse). It is not (currently) possible to vary the number of vertices dynamically.

Appearance

How should the stimulus look? Colour, borders, etc.

fill color [color] See *Color spaces*

color space [rgb, dkl, lms, hsv] See *Color spaces*

border color [color] See *Color spaces*

line width [int | float] How wide should the line be? Width is specified in chosen spatial units, see *Units for the window and stimuli*

opacity : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

ori [degrees] The orientation of the entire patch (texture and mask) in degrees.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

size [(width, height)] Size of the stimulus on screen

spatial units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

Texture

Control how the stimulus handles textures.

interpolate [linear, nearest] Should textures be interpolated?

units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

See also:

API reference for Polygon API reference for Rect API reference for ShapeStim #for arbitrary vertices

Pump Component

This component allows you to deliver liquid stimuli using a Cetoni neMESYS syringe pump.

Please specify the name of the pump configuration to use in the preferences under Hardware / Qmix pump configuration. See the [readme file](#) of the `pyqmix` project for details on how to set up your computer and create the configuration file.

Properties

Name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the stimulus should first appear.

Stop : Governs the duration for which the stimulus is presented.

Sync to screen [bool] Whether to synchronize the pump operations (starting, stopping) to the screen refresh. This ensures better synchronization with visual stimuli.

Hardware

Parameters for controlling hardware.

Pump index [int] The index of the pump: The first pump's index is 0, the second pump's index is 1, etc. You may insert the name of a variable here to adjust this value dynamically.

Syringe type [select the appropriate option] Currently, 25 mL and 50 mL glass syringes are supported. This setting ensures that the pump will operate at the correct flow rate.

Pump action [aspirate or dispense] Whether to fill (aspirate) or to empty (dispense) the syringe.

Flow rate [float] The flow rate in the selected flow rate units.

Flow rate unit [mL/s or mL/min] The unit in which the flow rate values are supplied.

Switch valve after dosing [bool] Whether to switch the valve position after the pump operation has finished. This can be used to ensure a sharp(er) stimulus offset.

RatingScale Component

The Rating Scale Component is in the process of deprecation, if possible we recommend using the newer Slider component instead. By combining a Slider, Text/TextBox and Button components, a Slider should be able to perform all the same tasks as a RatingScale Component.

A rating scale is used to collect a numeric rating or a choice from a few alternatives, via the mouse, the keyboard, or both. Both the response and time taken to make it are returned.

A given routine might involve an image (patch component), along with a rating scale to collect the response. A routine from a personality questionnaire could have text plus a rating scale.

Three common usage styles are enabled on the first settings page: ‘visual analog scale’: the subject uses the mouse to position a marker on an unmarked line

‘category choices’: choose among verbal labels (categories, e.g., “True, False” or “Yes, No, Not sure”)

‘scale description’: used for numeric choices, e.g., 1 to 7 rating

Complete control over the display options is available as an advanced setting, ‘customize_everything’.

Properties

name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : The duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

category choices [string] Instead of a numeric scale, you can present the subject with words or phrases to choose from. Enter all the words as a string. (Probably more than 6 or so will not look so great on the screen.) Spaces are assumed to separate the words. If there are any commas, the string will be interpreted as a list of words or phrases (possibly including spaces) that are separated by commas.

scaleDescription [string] Brief instructions, reminding the subject how to interpret the numerical scale, default = “1 = not at all ... extremely = 7”

forceEndRoutine [bool] If checked, when the subject makes a rating the routine will be ended.

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

size [float] The size controls how big the scale will appear on the screen. (Same as “displaySizeFactor”.) Larger than 1 will be larger than the default, smaller than 1 will be smaller than the default.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window. Default is centered left-right, and somewhat lower than the vertical center (0, -0.4).

Data

What information to save, how to lay it out and when to save it.

visualAnalogScale [checkbox] If this is checked, a line with no tick marks will be presented using the ‘glow’ marker, and will return a rating from 0.00 to 1.00 (quasi-continuous). This is intended to bias people away from thinking in terms of numbers, and focus more on the visual bar when making their rating. This supersedes either choices or scaleDescription.

low [str] The lowest number (bottom end of the scale), default = 1. If it’s not an integer, it will be converted to lowAnchorText (see Advanced).

high [str] The highest number (top end of the scale), default = 7. If it’s not an integer, it will be converted to highAnchorText (see Advanced).

labels [str] What labels should be applied

marker start : Where should the marker start at

store history [bool] Store full record of how participant moved on the slider

store rating [bool] Save the rating that was selected

store rating time [bool] Save the time from the beginning of the trial until the participant responds.

See also:

API reference for *RatingScale*

Resource Manager Component

(added version 2022.1.0)

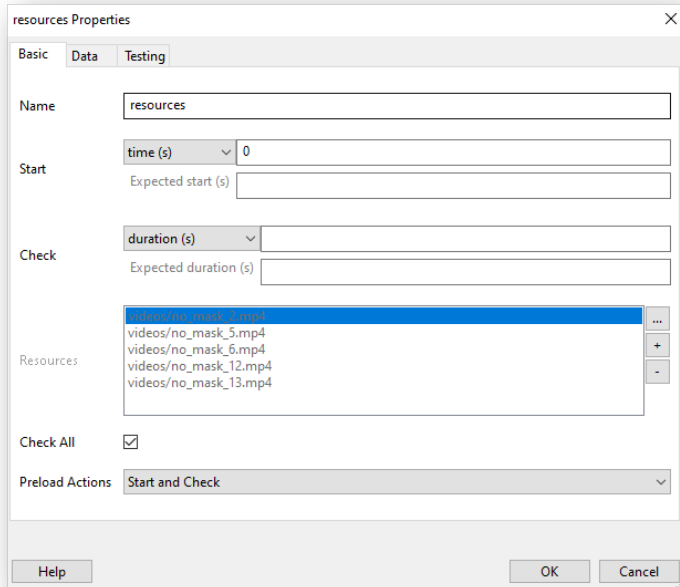
Pre-load resources into memory so that components using them can start without having to load first.

This component is only relevant to online studies. If you use this component it will override loading of resources at the beginning of the experiment. This means that any resources specified in the Experiment Settings > Online > Additional Resources will not be used, make sure to load all required resources within the experiment.

Most experiments need “resources” in order to run. Be it images, sounds, spreadsheets or movies. When running a study online through pavlovia.org, these resources are loaded by default at the beginning of the experiment, and you will usually see a loading bar.

However, sometimes this loading can take a pretty long time. This happens either because you have a very large number of resources or because individual files are large (e.g. long movies) . In cases like this, it may be preferred to load these within your experiment, for example whilst your participants are reading through the instructions, in an inter-trial interval or during a break between blocks. This is where the *Resource Manager* component and/or the *Static Component* come in.

You can find the Resource Manager under “Custom” in the Component Panel. The component has many properties similar to any other component, a start time, a duration etc. The most important fields in the component are **Resources**, indicating the list of resources to load, and **Preload Actions**, indicating if we are initiating loading (Start), checking previously initiated loading has completed (Check), or both (Start and Check). For experiments where we might have several resource manager components, we can also check if the resources from *all* components currently exist in memory by selecting “Check All”.



Example: Loading resources in the background of instructions

A common use case for resource manager might be to load resources in the background of instructions (or any routine!), and only let your participants move forward when the resources are loaded. To do this:

1. Add a resource manager component.
2. Populate the resources field with the resources to be loaded.
3. Set *Preload Actions* to *Start and Check*.
4. Add a code component and use this in the “Each Frame” tab (where “resources” refers to the name of your resource manager component):

```
if resources.status == FINISHED:
    continueRoutine = False
```

5. Alternatively to step 4, you might want to have an image or text that is clickable, but have *Start* set to `resources.status == FINISHED`. This will make the button “pop-up” when the resources have finished loading!

Note: The resource manager has an attribute “status” and we can check if it has finished using `resources.status == FINISHED` (where *resources* corresponds to the name of your resource manager component).

Loading resources for blocked or branched designs, or loading trial-by-trial

Sometimes we might have a design where participants only need to be presented with a subset of resources. We might have 100 movies, but group 1 sees 50 movies and group 2 sees the other 50. In cases like this you might ask “How to I make the resources in my resource manager conditional?”. Well, for designs like this we actually recommend you use something a little different, the *Static Component* - so check it out!

Reward Component

This component allows you to deliver a water reward to smalls animals (mice, rat) and monitor licks, using the peristaltic from Labeo Technologies Inc.

Properties

Name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the first pulse of the water reward occurs.

Stop : Governs the duration for which the water reward sequence is given.

Sync to screen [bool] Choose to synchronize the reward operations (pulses) to the screen refresh. This ensures better synchronization with visual stimuli.

Pulse duration (s) : The duration of the pulse sent to the peristaltic pump. To know exactly the volume of water given, please perform a calibration curve, as the quantity depends of your experimental setup. Precision can go as low as 16.6 ms for a 60 Hz screen.

Number of pulses : The number of pulses in a burst.

Delay between sequences (s) : A sequence is a burst sequence, so multiples pulses. This delay is the duration between these sequences

Number of sequences : The number of burst sequences occurring during the time the water reward component is enable in the experiment.

Delay between pulses (s) : The time duration between pulses in a burst sequence.

Save actions of pump and licks to txt file : Check if you want to save log events (pump ON, OFF, lick) to a .txt file, located in the data folder of the experiment.

COM port : Please specify the COM port (USB port) on which the pump is connected.

Serial Port Out Component

This component allows you to send triggers to a serial port. For a full tutorial please see :ref: *this page <serial>*.

An example usage would be in EEG experiments to set the port to 0 when no stimuli are present and then set it to an identifier value for each stimulus synchronised to the start/stop of that stimulus. In that case you might set the *Start data* to be *\$ID* (with ID being a column in your conditions file) and set the *Stop Data* to be “0”.

Properties

Name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

Stop : Governs the duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

Port address [type the appropriate option] You need to know the address of the serial port you wish to write to. For more information on how to find out this address please see :ref: *this page <serial>*.

Start data [string] When the start time/condition occurs this value will be sent to the serial port. For more information please see :ref: *this page <serial>*.

Stop data [string] As with start data but sent at the end of the period.

Data

Sync timing with screen refresh [boolean] If true then the serial port will be sent synchronised to the next screen refresh, which is ideal if it should indicate the onset of a visual stimulus. If set to False then the data will be sent on the serial port immediately.

Get response? [boolean] If true then PsychoPy reads and records a response from the port after the data has been sent.

Hardware

Parameters for controlling hardware.

Baud rate : The baud rate, or speed, of the connection.

Data bits : The size of the bits to be sent.

Stop bits : The size of the bits to be sent on stop.

Parity : The parity mode.

Timeout : Time at which to give up listening for a response from the serial port.

Slider Component

A Slider uses mouse input to collect ratings, all sliders have the same basic structure (a line, rectangle or series of dots to indicate the range of values, several tick marks and labels, a marker) but their appearance can be varied significantly by changing the *style* parameter. For example, a *radio* style Slider features several dots and a circular marker, while a *scrollbar* style Slider features a translucent rectangle with a long marker like the page scrollbar on a website.

Properties

Name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

Start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

Stop : The duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

Size [(width, height)] The size controls the width and height of the slider. The slider is oriented horizontally when the width is greater than the height, and oriented vertically otherwise. Default is (1.0, 0.1)

Position [(X,Y)] The position of the centre of the stimulus, in the units specified by the stimulus or window. Default is centered left-right, and somewhat lower than the vertical center (0, -0.4).

Ticks [(list or tuple of integers)] The ticks that will be placed on the slider scale. The first and last ticks will be placed on the ends of the slider, and the remaining are spaced between the endpoints corresponding to their values. For example, (1, 2, 3, 4, 5) will create 5 evenly spaced ticks. (1, 3, 5) will create three ticks, one at each end and one in the middle.

Labels [(list or tuple of strings)] The text to go with each tick (or spaced evenly across the ticks). If you give 3 labels but 5 tick locations then the end and middle ticks will be given labels. If the labels can't be distributed across the ticks then an error will be raised. If you want an uneven distribution you should include a list matching the length of ticks but with some values set to None.

Granularity : Specifies step size for rating. 0 corresponds to a continuous scale, 1 corresponds to an integer or discrete scale.

Force end of Routine : If checked, when the subject makes a rating the routine will be ended.

Opacity [value from 0 to 1 or None] Setting the opacity of the Slider will set the opacity of all of its parts to the same value, to control these individually set opacity to None

Units : See *Units for the window and stimuli*.

Flip : By default labels are below the scale or left of the scale. By checking this checkbox, the labels are placed above the scale or to the right of the scale.

Formatting

Font : Font for labels.

Letter Height : Font size for labels,

Appearance

Label Color : Color of the labels. See *Color spaces*.

Marker Color : Color of the marker. See *Color spaces*.

Line Color : Color of the lines or, for styles such as *scrollbar* or *slider*, the backboard. See *Color spaces*.

Styles : A selection of pre-defined styles.

Style Tweaks : Tweaks to the style of the slider which can be applied on top of the overall style - multiple tweaks can be selected.

See also:

API reference for *Slider*

Sound Component

Parameters

name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start [float or integer] The time that the stimulus should first play. See *Defining the onset/duration of components* for details.

stop : For sounds loaded from a file leave this blank and then give the *Expected duration* below for visualisation purposes. See *Defining the onset/duration of components* for details.

sound : This sound can be described in a variety of ways:

- a number can specify the frequency in Hz (e.g. 440)
- a letter gives a note name (e.g. “C”) and sharp or flat can also be added (e.g. “Csh” “Bf”)
- a filename, which can be a relative or absolute path (mid, wav, and ogg are supported).

Playback

How should stimulus play? Speed, volume, etc.

volume [float or integer] The volume with which the sound should be played. It’s a normalized value between 0 (minimum) and 1 (maximum).

hamming window [bool] Should there be a hamming window between stimuli?

See also:

API reference for `SoundPyo`

Static Component

(Added in Version 1.78.00. Made compatible for online use version 2022.1)

The Static Component allows you to have a period where you can preload images or perform other time-consuming operations that not be possible while the screen is being updated. Static periods are also particularly useful for *online* studies to decrease the time taken to load resources at the start (see also *Resource Manager Component*).

Note: For online studies, if you use a static component this will override the resources loaded at the beginning via Experiment settings > Online > Additional resources. You might therefore want to combine a static period with a *Resource Manager Component* to make sure that all resources your study needs will be loaded and available for the experiment.

Typically a static period would be something like an inter-trial or inter-stimulus interval (ITI/ISI). During this period you should not have any other objects being presented that are being updated (this isn’t checked for you - you have to make that check yourself), but you can have components being presented that are themselves static. For instance a fixation point never changes and so it can be presented during the static period (it will be presented and left on-screen while the other updates are being made).

Any stimulus updates can be made to occur during any static period defined in the experiment (it does not have to be in the same Routine). This is done in the updates selection box- once a static period exists it will show up here as well as the standard options of *constant* and *every repeat* etc. Many parameter updates (e.g. orientation are made so

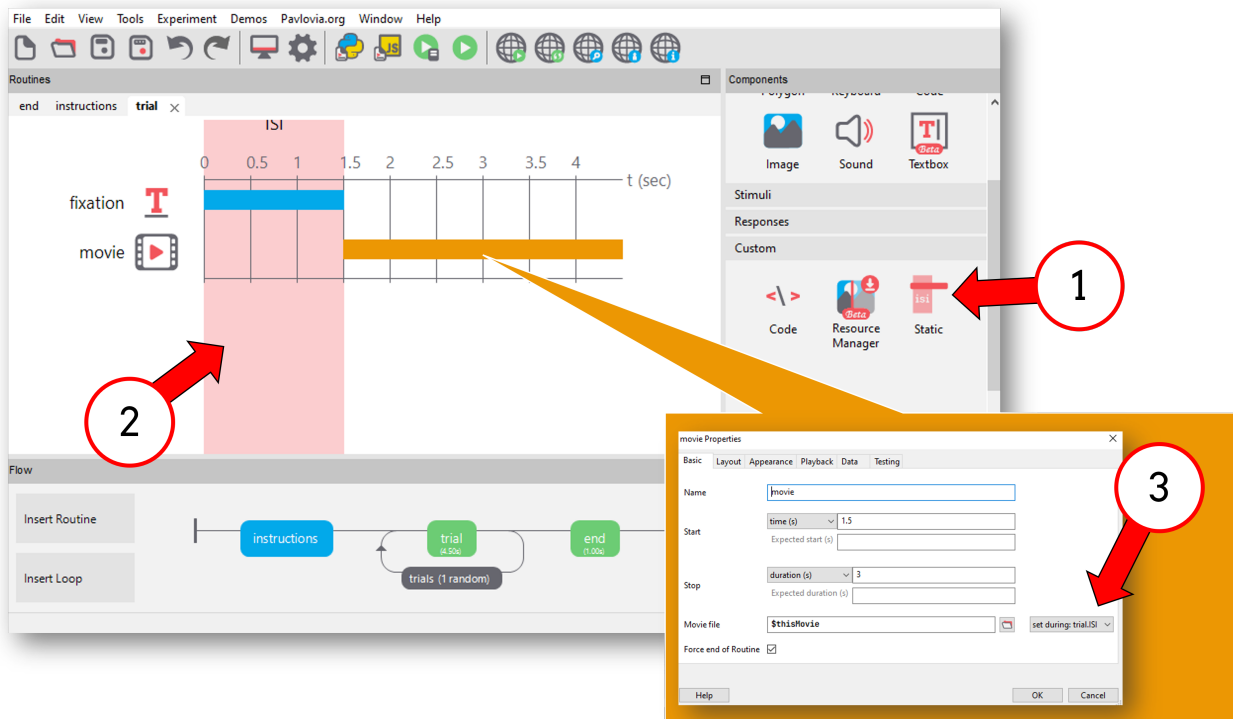


Fig. 5.5: How to use a static component. 1) To use a static component first select it from the component panel. 2) highlights in red the time window you are treating as “static”. If you click on the red highlighted window you can edit the static component. 3) To use the static window to load a resource, select the component where the resource will be load, and in the dropdown window choose “set during:trial.ISI” - here “trial” refers to the routine where the static component is and “ISI” refers to the name of the static component.

quickly that using the static period is of no benefit but others, most notably the loading of images from disk, can take substantial periods of time and these should always be performed during a static period to ensure good timing.

If the updates that have been requested were not completed by the end of the static period (i.e. there was a timing overshoot) then you will receive a warning to that effect. In this case you either need a longer static period to perform the actions or you need to reduce the time required for the action (e.g. use an image with fewer pixels).

Parameters

name : Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the static period begins. See *Defining the onset/duration of components* for details.

stop : The time that the static period ends. See *Defining the onset/duration of components* for details.

Custom

Parameters for injecting custom code

custom code : After running the component updates (which are defined in each component, not here) any code inserted here will also be run

See also:

API reference for *StaticPeriod*

Text Component

This component can be used to present text to the participant, either instructions or stimuli.

name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : The duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

text [string] Text to be shown

Appearance

How should the stimulus look? Colour, borders, etc.

foreground color : See *Color spaces*

foreground color space [rgb, dkl or lms] See *Color spaces*

opacity : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

flip : Whether to mirror-reverse the text: ‘horiz’ for left-right mirroring, ‘vert’ for up-down mirroring. The flip can be set dynamically on a per-frame basis by using a variable, e.g., \$mirror, as defined in a code component or conditions file and set to either ‘horiz’ or ‘vert’.

ori [degrees] The orientation of the stimulus in degrees.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

spatial units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

wrap width [code] How many characters in should text be wrapped at?

Formatting

Formatting text

font [string] What font should the text be set in? Must be the name of a font installed on your computer

language style [LTR, RTL, Arabic] Should text be laid out from left to right (LTR), from right to left (RTL), or laid out like Arabic script?

letter height [integer or float] The height of the characters in the given units of the stimulus/window. Note that nearly all actual letters will occupy a smaller space than this, depending on font, character, presence of accents etc. The width of the letters is determined by the aspect ratio of the font.

See also:

API reference for *TextStim*

Textbox Component

This component can be used either to present text to the participant, or to allow free-text answers via the keyboard.

name [string] Everything in a experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces).

start : The time that the stimulus should first appear. See *Defining the onset/duration of components* for details.

stop : The duration for which the stimulus is presented. See *Defining the onset/duration of components* for details.

editable [bool] Whether this Textbox can be edited by the participant (text input) or not (static text).

text [string] Text to be shown

Appearance

How should the stimulus look? Colour, borders, etc.

text color [color] See *Color spaces*

fill color [color] See *Color spaces*

border color [color] See *Color spaces*

color space [rgb, dkl, lms, hsv] See *Color spaces*

border width [int | float] How wide should the line be? Width is specified in chosen spatial units, see *Units for the window and stimuli*

opacity : Vary the transparency, from 0.0 = invisible to 1.0 = opaque

Layout

How should the stimulus be laid out? Padding, margins, size, position, etc.

flip horizontal [bool] Whether to mirror-reverse the text horizontally (left-right mirroring)

flip vertical [bool] Whether to mirror-reverse the text vertically (top-bottom mirroring)

ori [degrees] The orientation of the stimulus in degrees.

pos [[X,Y]] The position of the centre of the stimulus, in the units specified by the stimulus or window

size [(width, height)] Size of the stimulus on screen

spatial units [deg, cm, pix, norm, or inherit from window] See *Units for the window and stimuli*

padding [float] How much space should there be between the box edge and the text?

anchor [center, center-left, center-right, top-left, top-center, top-right, bottom-left, bottom-center, bottom-right] What point on the textbox should be anchored to its position? For example, if the position of the TextBox is (0, 0), should the middle of the textbox be in the middle of the screen, should its top left corner be in the middle of the screen, etc.?

Formatting

Formatting text

font [string] What font should the text be set in? Can be a font installed on your computer, saved to the “fonts” folder in your user folder or (if you are connected to the internet), a font from Google Fonts.

language style [LTR, RTL, Arabic] Should text be laid out from left to right (LTR), from right to left (RTL), or laid out like Arabic script?

letter height [integer or float] The height of the characters in the given units of the stimulus/window. Note that nearly all actual letters will occupy a smaller space than this, depending on font, character, presence of accents etc. The width of the letters is determined by the aspect ratio of the font.

line spacing [float] How tall should each line be, proportional to the size of the font?

See also:

API reference for *TextBox*

Variable Component

A variable can hold quantities or values in memory that can be referenced using a variable name. You can store values in a variable to use in your experiments.

Parameters

Name [string] Everything in an experiment needs a unique name. The name should contain only letters, numbers and underscores (no punctuation marks or spaces). The variable name references the value stored in memory, so that your stored values can be used in your experiments.

Start [int, float or bool] The time or condition from when you want your variable to be defined. The default value is None, and so will be defined at the beginning of the experiment, trial or frame. See *Defining the onset/duration of components* for details.

Stop [int, float or bool] The duration for which the variable is defined/updated. See *Defining the onset/duration of components* for details.

Experiment start value: any The variable can take any value at the beginning of the experiment, so long as you define your variables using literals or existing variables.

Routine start value [any] The variable can take any value at the beginning of a routine/trial, and can remain a constant, or be defined/updated on every routine.

Frame start value [any] The variable can take any value at the beginning of a frame, or during a condition based on Start and/or Stop.

Data

What information to save, how to lay it out and when to save it.

Save exp start value [bool] Choose whether or not to save the experiment start value to your data file.

Save routine start value [bool] Choose whether or not to save the routine start value to your data file.

Save frame value [bool and drop=down menu] Frame values are contained within a list for each trial, and discarded at the end of each trial. Choose whether or not to take the first, last or average variable values from the frame container, and save to your data file.

Save routine end value [bool] Choose whether or not to save the routine end value to your data file.

Save exp end value [bool] Choose whether or not to save the experiment end value to your data file.

Entering parameters

Most of the entry boxes for Component parameters simply receive text or numeric values or lists (sequences of values surrounded by square brackets) as input. In addition, the user can insert variables and code into most of these, which will be interpreted either at the beginning of the experiment or at regular intervals within it.

To indicate to that the value represents a variable or python code, rather than literal text, it should be preceded by a \$. For example, inserting *intensity* into the text field of the Text Component will cause that word literally to be presented, whereas *\$intensity* will cause python to search for the variable called intensity in the script.

Variables associated with *Loops* can also be entered in this way (see *Using loops to update stimuli trial-by-trial* for further details). But it can also be used to evaluate arbitrary python code.

For example:

- `$random(2)` will generate a pair of random numbers
- `["$yn"][randint(2)]` will randomly choose the first or second character (y or n)
- `$globalClock.getTime()` will insert the current time in secs of the globalClock object

- $[\sin(\text{angle}), \cos(\text{angle})]$ will insert the sin and cos of an angle (e.g. into the x,y coords of a stimulus)

How often to evaluate the variable/code

If you do want the parameters of a stimulus to be evaluated by code in this way you need also to decide how often it should be updated. By default, the parameters of Components are set to be *constant*; the parameter will be set at the beginning of the experiment and will remain that way for the duration. Alternatively, they can be set to change either on *every repeat* in which case the parameter will be set at the beginning of the Routine on each repeat of it. Lastly many parameters can even be set *on every frame*, allowing them to change constantly on every refresh of the screen.

5.1.6 Experiment settings

The settings menu can be accessed by clicking the icon at the top of the window. It allows the user to set various aspects of the experiment, such as the size of the window to be used or what information is gathered about the subject and determine what outputs (data files) will be generated.

Settings

Basic

Experiment name A name that will be stored in the metadata of the data file.

Use PsychoPy version Which version of was the task created in? if you are using a more recently installed version of this can compile using an archived, older version to run previously created tasks.

Show info dlg If this box is checked then a dialog will appear at the beginning of the experiment allowing the *Experiment Info* to be changed.

Enable escape If ticked then the *Esc* key can be used to exit the experiment at any time (even without a keyboard component)

Experiment Info This information will be presented in a dialog box at the start and will be saved with any data files and so can be used for storing information about the current run of the study. The information stored here can also be used within the experiment. For example, if the *Experiment Info* included a field called *ori* then Builder *Components* could access `expInfo['ori']` to retrieve the orientation set here. Obviously this is a useful way to run essentially the same experiment, but with different conditions set at run-time. If you are running a study online, we recommend keeping the field “participant” because this is used to name data output files.

Screen

Monitor The name of the monitor calibration. Must match one of the monitor names from *Monitor Center*.

Screen: If multiple screens are available (and if the graphics card is *not* an intel integrated graphics chip) then the user can choose which screen they use (e.g. 1 or 2).

Full-screen window: If this box is checked then the experiment window will fill the screen (overriding the window size setting and using the size that the screen is currently set to in the operating system settings).

Window size: The size of the window in pixels, if this is not to be a full-screen window.

Units The default units of the window (see *Units for the window and stimuli*). These can be overridden by individual *Components*.

Audio

Audio library Choice of audio library to use to present sound, default uses preferences (see *Preferences*).

Audio latency priority Latency mode for PsychToolbox audio (see *Preferences*) (because this applies to the PTB sound backend, this only applies for local, not online studies)

Force stereo Force audio to stereo (2-channel) output

Online

Output path Where to export the compiled javascript experiment and associated html files. (note that in earlier versions of this was *html* by default, this is not necessary as it will duplicate your resources, associated discourse threads with this suggestion might now be outdated)

Export html When to export a html file and compile a javascript version of the experiment. This is on sync by default, meaning these files will be generated when a project is pushed/synced to . Alternatively this can be “on save” or “manually” the latter might be used if you are making manual edits to the exported javascript file, though this is not recommended as changes will not be reflected back in your builder file.

Completed URL The URL to direct participants to upon completion (when they select “OK” in the green thank-you message online)

Incomplete URL The URL to direct participants to if they exit the task early (e.g. by pressing the escape key).

Additional resources Resources that your task will require (e.g. image files, excel sheets). Note that will attempt to populate this automatically, though if you encounter an “Unknown resource” error online, it is possible that you need to add resources to this list.

Eyetracking

Eyetracker Device Specify what kind of eye tracker you are using. If you are creating your paradigm out-of-lab (i.e. with no eye tracker) we suggest using MouseGaze, which will use your mouse to simulate eye movements and blinks. Alternatively, you can select which device you are currently using and set-up those parameters (see *ioHub Common Eye Tracker Interface*)

Data

Data filename: A *formatted string* to control the base filename and path, often based on variables such as the date and/or the participant. This base filename will be given the various extensions for the different file types as needed. Examples:

```
# all in data folder relative to experiment file: data/JWP_memoryTask_2014_Feb_15_
→1648
'data/%s_%s_%s' %(expInfo['participant'], expName, expInfo['date'])

# group by participant folder: data/JWP/memoryTask-2014_Feb_15_1648
'data/%s/%s-%s' %(expInfo['participant'], expName, expInfo['date'])

# put into dropbox: ~/dropbox/data/memoryTask/JWP-2014_Feb_15_1648
# os.path.expanduser replaces '~' with the path to your home directory,
# os.path.join joins the path components together correctly, regardless of OS
```

(continues on next page)

(continued from previous page)

```
# os.path.relpath creates a relative path between the specified path and the
↳current directory
'$os.path.relpath(os.path.join(os.path.expanduser('~'), 'dropbox', 'data',
↳expName, expInfo['participant'] + '-' + expInfo['date']))
```

Data file delimiter What delimiter should your data file use to separate the columns

Save Excel file If this box is checked an Excel data file (.xlsx) will be stored.

Save csv file (summaries) If this box is checked a summary file will be created with one row corresponding to the entire loop. If a keyboard response is used the mean and standard deviations of responses across trials will also be stored.

Save csv file (trial-by-trial) If this box is checked a comma separated variable (.csv) will be stored. Each trial will be stored as a new row.

Save psydat file If this box is checked a *data file (.psydat)* will be stored. This is a Python specific format (.pickle files) which contains more information than .xlsx or .csv files that can be used with data analysis and plotting scripts written in Python. Whilst you may not wish to use this format it is recommended that you always save a copy as it contains a complete record of the experiment at the time of data collection.

Save hdf5 file If this box is checked data will be stored to a hdf5 file, this is mainly applicable if a component is implemented that requires a complex data structure e.g. eye-tracking.

Save log file A log file provides a record of what occurred during the experiment in chronological order, including information about any errors or warnings that may have occurred.

Logging level How much detail do you want to be output to the log file, if it is being saved. The lowest level is *error*, which only outputs error messages; *warning* outputs warnings and errors; *info* outputs all info, warnings and errors; *debug* outputs all info that can be logged. This system enables the user to get a great deal of information while generating their experiments, but then reducing this easily to just the critical information needed when actually running the study. If your experiment is not behaving as you expect it to, this is an excellent place to begin to work out what the problem is.

5.1.7 Defining the onset/duration of components

As of version 1.70.00, the onset and offset times of stimuli can be defined in several ways.

Start and stop times can be entered in terms of seconds (*time (s)*), by frame number (*frameN*) or in relation to another stimulus (*condition*). *Condition* would be used to make *Components* start or stop depending on the status of something else, for example when a sound has finished. Duration can also be varied using a *Code Component*.

If you need very precise timing (particularly for very brief stimuli for instance) then it is best to control your onset/duration by specifying the number of frames the stimulus will be presented for.

Measuring duration in seconds (or milliseconds) is not very precise because it doesn't take into account the fact that your monitor has a fixed frame rate. For example if the screen has a refresh rate of 60Hz you cannot present your stimulus for 120ms; the frame rate would limit you to 116.7ms (7 frames) or 133.3ms (8 frames). The duration of a frame (in seconds) is simply 1/refresh rate in Hz.

Condition would be used to make *Components* start or stop depending on the status of something else, for example when a movie has finished. Duration can also be varied using a code component.

In cases where cannot determine the start/endpoint of your Component (e.g. because it is a variable) you can enter an 'Expected' start/duration. This simply allows components with variable durations to be drawn in the Routine window. If you do not enter the approximate duration it will not be drawn, but this will not affect experimental performance.

For more details of how to achieve good temporal precision see *Timing Issues and synchronisation*

Examples

- Use *time(s)* or *frameN* and simply enter numeric values into the start and duration boxes.
- Use *time(s)* or *frameN* and enter a numeric value into the start time and set the duration to a variable name *by preceding it with a \$*. Then set *expected time* to see an approximation in your *routine*
- Use condition to cause the stimulus to start immediately after a movie component called myMovie, by entering *\$myMovie.status==FINISHED* into the *start time*.

5.1.8 Generating outputs (datafiles)

There are 4 main forms of *output file* from :

- Excel 2007 files (.xlsx) see *Excel Data Files* for more details
- text data files (.csv, .tsv, or .txt) see *Delimited Text Files* for more details
- binary data files (.psydat) see *PsychoPy Data Files* for more details
- log files (.log) see *Log Files* for more details

5.1.9 Common Mistakes (aka Gotcha's)

General Advice

- Python and therefore is CASE SENSITIVE
- To use a dollar sign (\$) for anything other than to indicate a code snippet for example in a text, precede it with a backslash \\$ (the backslash won't be printed)
- Have you entered your the settings for your *monitor*? If you are using degrees as a unit of measurement and have not entered your monitor settings, the size of stimuli will not be accurate.
- If your experiment is not behaving in the way that you expect. Have you looked at the *log file*? This can point you in the right direction. Did you know you can change the type of information that is stored in the log file in preferences by changing the *logging level*.
- Have you tried compiling the script and running it. Does this produce a particular error message that points you at a particular problem area? You can also change things in a more detailed way in the coder view and if you are having problems, reading through the script can highlight problems. Reading a compiled script can also help with the creation of a *Code Component*

My stimulus isn't appearing, there's only the grey background

- Have you checked the size of your stimulus? If it is 0.5x0.5 pixels you won't be able to see it!
- Have you checked the position of your stimulus? Is it positioned off the screen?

The loop isn't using my Excel spreadsheet

- Have you remembered to specify the file you want to use when setting up the loop?
- Have you remembered to add the variables preceded by the \$ symbol to your stimuli?

I just want a plain square, but it's turning into a grating

- If you don't want your stimulus to have a texture, you need Image to be None

The code snippet I've entered doesn't do anything

- Have you remembered to put a \$ symbol at the beginning (this isn't necessary, and should be avoided in a *Code Component*)?
- A dollar sign as the first character of a line indicates to that the rest of the line is code. It does not indicate a variable name (unlike in perl or php). This means that if you are, for example, using variables to determine position, enter $[x,y]$. The temptation is to use $[\$x,\$y]$, which will not work.

My stimulus isn't changing as I progress through the loop

- Have you changed the setting for the variable that you want to change to 'change every repeat' (or 'change every frame')?

I'm getting the error message AttributeError: 'unicode object has no attribute 'XXXX''

- This type of error is usually caused by a naming conflict. Whilst we have made every attempt to make sure that these conflicts produce a warning message it is possible that they may still occur.
- The most common source of naming conflicts in an external file which has been imported to be used in a loop i.e. .xlsx, .csv.
- Check to make sure that all of the variable names are unique. There can be no repeated variable names anywhere in your experiment.

The window opens and immediately closes

- Have you checked all of your variable entries are accepted commands e.g. gauss but not Gauss
- If you compile your experiment and run it from the coder window what does the error message say? Does it point you towards a particular variable which may be incorrectly formatted?

If you are having problems getting the application to run please see [Troubleshooting](#)

5.1.10 Compiling a Script

If you click the *compile script* icon this will display the script for your experiment in the *Coder* window.

This can be used for debugging experiments, entering small amounts of code and learning a bit about writing scripts amongst other things.

The code is fully commented and so this can be an excellent introduction to writing your own code.

5.1.11 Set up your monitor properly

It's a really good idea to tell about the set up of your monitor, especially the size in cm and pixels and its distance, so that can present your stimuli in units that will be consistent in another lab with a different set up (e.g. cm or degrees of visual angle).

You should do this in *Monitor Center* which can be opened from Builder by clicking on the icon that shows two monitors. In *Monitor Center* you can create settings for multiple configurations, e.g. different viewing distances or different physical devices and then select the appropriate one by name in your experiments or scripts.

Having set up your monitor settings you should then tell which of your monitor setups to use for this experiment by going to the *Experiment settings* dialog.

CODER

The coder view is designed for those wishing to make scripts from scratch, either to make their experiments or do other things. Coder view does not teach you about Python *per se*, and you are recommended also to learn about that (Python has many excellent tutorials for programmers and non-programmers alike). In particular, dictionaries, lists and numpy arrays are used a great deal in most experiments.

You can program experiments in any python development environment (e.g. [PyCharm](#), [Spyder](#) would be excellent examples of full-featured editors). So, *why use Coder view in PsychoPy?* The answer is that the PsychoPy as a standalone package also includes several common python libraries you would use when making experiments in python. In general there will therefore be fewer steps to take to configure your python environment in coder. So if you are teaching python, there should be less work to set up the environment for each student! *However* if you are teaching python for many purposes beyond making experiments, you might want to move to another IDE (Integrated Development Environment), because coder won't have *everything* you need imported.

You can learn to use the scripting interface to in several ways, and you should probably follow a combination of them:

- Check the content of our [PsychoPy workshops](#) (we currently focus on coding concepts on day 3).
- *Basic Concepts*: some of the logic of scripting
- *Tutorials*: walk you through the development of some semi-complete experiments
- demos: in the demos menu of Coder view.
- use the *Builder* to *compile a script* and see how it works (you can actually compile to Python or Javascript to learn a bit of both!). This is also useful for understanding the *Code Component*, you can write a snippet in a code component in builder and compile to see where it is written in the script (but remember exporting to coder is a one way street, you can't make edits in coder and hope that is reflected back in the builder experiment).

You should check the [Reference Manual \(API\)](#) for further details and, ultimately, go into [PsychoPy](#) and start examining the source code. It's just regular python!

6.1 Basic Concepts

6.1.1 Presenting Stimuli

Note: Before you start, tell about your monitor(s) using the *Monitor Center*. That way you get to use units (like degrees of visual angle) that will transfer easily to other computers.

Stimulus objects

Python is an ‘object-oriented’ programming language, meaning that most stimuli in are represented by python objects, with various associated methods and information.

Typically you should create your stimulus with the initial desired attributes once, at the beginning of the script, and then change select attributes later (see section below on setting stimulus attributes). For instance, create your text and then change its color any time you like:

```
from psychopy import visual, core
win = visual.Window([400,400])
message = visual.TextStim(win, text='hello')
message.autoDraw = True # Automatically draw every frame
win.flip()
core.wait(2.0)
message.text = 'world' # Change properties of existing stim
win.flip()
core.wait(2.0)
```

Setting stimulus attributes

Stimulus attributes are typically set using either:

- a string, which is just some characters (as `message.text = 'world'` above)
- a scalar (a number; see below)
- an x,y-pair (two numbers; see below)

x,y-pair:

is very flexible in terms of input. You can specify the widely used x,y-pairs using these types:

- A Tuple (x, y) with two elements
- A List [x, y] with two elements
- A numpy array([x, y]) with two elements

However, always converts the x,y-pairs to numpy arrays internally. For example, all three assignments of pos are equivalent here:

```
stim.pos = (0.5, -0.2) # Right and a bit up from the center
print(stim.pos) # array([0.5, -0.2])

stim.pos = [0.5, -0.2]
print(stim.pos) # array([0.5, -0.2])

stim.pos = numpy.array([0.5, -0.2])
print(stim.pos) # array([0.5, -0.2])
```

Choose your favorite :-) However, you can't assign elementwise:

```
stim.pos[1] = 4 # has no effect
```

Scalar: Int or Float.

Mostly, scalars are no-brainers to understand. E.g.:

```
stim.ori = 90 # Rotate stimulus 90 degrees
stim.opacity = 0.8 # Make the stimulus slightly transparent.
```

However, scalars can also be used to assign x,y-pairs. In that case, both x and y get the value of the scalar. E.g.:

```
stim.size = 0.5
print(stim.size) # array([0.5, 0.5])
```

Operations on attributes: Operations during assignment of attributes are a handy way to smoothly alter the appearance of your stimuli in loops.

Most scalars and x,y-pairs support the basic operations:

```
stim.attribute += value # addition
stim.attribute -= value # subtraction
stim.attribute *= value # multiplication
stim.attribute /= value # division
stim.attribute %= value # modulus
stim.attribute **= value # power
```

They are easy to use and understand on scalars:

```
stim.ori = 5 # 5.0, set rotation
stim.ori += 3.8 # 8.8, rotate clockwise
stim.ori -= 0.8 # 8.0, rotate counterclockwise
stim.ori /= 2 # 4.0, home in on zero
stim.ori **= 3 # 64.0, exponential increase in rotation
stim.ori %= 10 # 4.0, modulus 10
```

However, they can also be used on x,y-pairs in very flexible ways. Here you can use both scalars and x,y-pairs as operators. In the latter case, the operations are element-wise:

```
stim.size = 5 # array([5.0, 5.0]), set quadratic size
stim.size += 2 # array([7.0, 7.0]), increase size
stim.size /= 2 # array([3.5, 3.5]), downscale size
stim.size += (0.5, 2.5) # array([4.0, 6.0]), a little wider and much taller
stim.size *= (2, 0.25) # array([8.0, 1.5]), upscale horizontal and downscale
↳vertical
```

Operations are not meaningful for strings.

Timing

There are various ways to measure and control timing in :

- using frame refresh periods (most accurate, least obvious)
- checking the time on Clock objects
- using `core.wait()` commands (most obvious, least flexible/accurate)

Using `core.wait()`, as in the above example, is clear and intuitive in your script. But it can't be used while something is changing. For more flexible timing, you could use a `Clock()` object from the `core` module:

```
from psychopy import visual, core

# Setup stimulus
```

(continues on next page)

(continued from previous page)

```

win = visual.Window([400, 400])
gabor = visual.GratingStim(win, tex='sin', mask='gauss', sf=5, name='gabor')
gabor.autoDraw = True # Automatically draw every frame
gabor.autoLog = False # Or we'll get many messages about phase change

# Let's draw a stimulus for 2s, drifting for middle 0.5s
clock = core.Clock()
while clock.getTime() < 2.0: # Clock times are in seconds
    if 0.5 <= clock.getTime() < 1.0:
        gabor.phase += 0.1 # Increment by 10th of cycle
    win.flip()

```

Clocks are accurate to around 1ms (better on some platforms), but using them to time stimuli is not very accurate because it fails to account for the fact that one frame on your monitor has a fixed frame rate. In the above, the stimulus does not actually get drawn for exactly 0.5s (500ms). If the screen is refreshing at 60Hz (16.7ms per frame) and the `getTime()` call reports that the time has reached 1.999s, then the stimulus will draw again for a frame, in accordance with the `while` loop statement and will ultimately be displayed for 2.0167s. Alternatively, if the time has reached 2.001s, there will not be an extra frame drawn. So using this method you get timing accurate to the nearest frame period but with little consistent precision. An error of 16.7ms might be acceptable to long-duration stimuli, but not to a brief presentation. It also might also give the false impression that a stimulus can be presented for any given period. At 60Hz refresh you can not present your stimulus for, say, 120ms; the frame period would limit you to a period of 116.7ms (7 frames) or 133.3ms (8 frames).

As a result, the most precise way to control stimulus timing is to present them for a specified number of frames. The frame rate is extremely precise, much better than ms-precision. Calls to `Window.flip()` will be synchronised to the frame refresh; the script will not continue until the flip has occurred. As a result, on most cards, as long as frames are not being 'dropped' (see *Detecting dropped frames*) you can present stimuli for a fixed, reproducible period.

Note: Some graphics cards, such as Intel GMA graphics chips under win32, don't support frame sync. Avoid integrated graphics for experiment computers wherever possible.

Using the concept of fixed frame periods and `flip()` calls that sync to those periods we can time stimulus presentation extremely precisely with the following:

```

from psychopy import visual, core

# Setup stimulus
win = visual.Window([400, 400])
gabor = visual.GratingStim(win, tex='sin', mask='gauss', sf=5,
    name='gabor', autoLog=False)
fixation = visual.GratingStim(win, tex=None, mask='gauss', sf=0, size=0.02,
    name='fixation', autoLog=False)

# Let's draw a stimulus for 200 frames, drifting for frames 50:100
for frameN in range(200): # For exactly 200 frames
    if 10 <= frameN < 150: # Present fixation for a subset of frames
        fixation.draw()
    if 50 <= frameN < 100: # Present stim for a different subset
        gabor.phase += 0.1 # Increment by 10th of cycle
        gabor.draw()
    win.flip()

```


Using autoDraw

Stimuli are typically drawn manually on every frame in which they are needed, using the `draw()` function. You can also set any stimulus to start drawing every frame using `stim.autoDraw = True` or `stim.autoDraw = False`. If you use these commands on stimuli that also have `autoLog=True`, then these functions will also generate a log message on the frame when the first drawing occurs and on the first frame when it is confirmed to have ended.

6.1.2 Logging data

`TrialHandler` and `StairHandler` can both generate data outputs in which responses are stored, in relation to the stimulus conditions. In addition to those data outputs, can create detailed chronological log files of events during the experiment.

Log levels and targets

Log messages have various levels of severity: ERROR, WARNING, DATA, EXP, INFO and DEBUG

Multiple *targets* can also be created to receive log messages. Each target has a particular critical level and receives all logged messages greater than that. For example, you could set the console (visual output) to receive only warnings and errors, have a central log file that you use to store warning messages across studies (with file mode *append*), and another to create a detailed log of data and events within a single study with *level=INFO*:

```
from psychopy import logging
logging.console.setLevel(logging.WARNING)
# overwrite (filemode='w') a detailed log of the last run in this dir
lastLog = logging.LogFile("lastRun.log", level=logging.INFO, filemode='w')
# also append warnings to a central log file
centralLog = logging.LogFile("C:\\psychopyExps.log", level=logging.WARNING, filemode=
    ↪ 'a')
```

Updating the logs

For performance purposes log files are not actually written when the log commands are ‘sent’. They are stored in a list and processed automatically when the script ends. You might also choose to force a *flush* of the logged messages manually during the experiment (e.g. during an inter-trial interval):

```
from psychopy import logging
...
logging.flush()    # write messages out to all targets
```

This should only be necessary if you want to see the logged information as the experiment progresses.

AutoLogging

New in version 1.63.00

Certain events will log themselves automatically by default. For instance, visual stimuli send log messages every time one of their parameters is changed, and when autoDraw is toggled they send a message that the stimulus has started/stopped. All such log messages are timestamped with the frame flip on which they take effect. To avoid this logging, for stimuli such as fixation points that might not be critical to your analyses, or for stimuli that change constantly and will flood the logging system with messages, the autoLogging can be turned on/off at initialisation of the stimulus and can be altered afterwards with `.setAutoLog(True/False)`

Manual methods

In addition to a variety of automatic logging messages, you can create your own, of various levels. These can be timestamped immediately:

```
from psychopy import logging
logging.log(level=logging.WARN, msg='something important')
logging.log(level=logging.EXP, msg='something about the conditions')
logging.log(level=logging.DATA, msg='something about a response')
logging.log(level=logging.INFO, msg='something less important')
```

There are additional convenience functions for the above: `logging.warn('a warning')` etc.

For stimulus changes you probably want the log message to be timestamped based on the frame flip (when the stimulus is next presented) rather than the time that the log message is sent:

```
from psychopy import logging, visual
win = visual.Window([400,400])
win.flip()
logging.log(level=logging.EXP, msg='sent immediately')
win.logOnFlip(level=logging.EXP, msg='sent on actual flip')
win.flip()
```

Using a custom clock for logs

New in version 1.63.00

By default times for log files are reported as seconds after the very beginning of the script (often it takes a few seconds to initialise and import all modules too). You can set the logging system to use any given `core.Clock` object (actually, anything with a `getTime()` method):

```
from psychopy import core, logging
globalClock = core.Clock()
logging.setDefaultClock(globalClock)
```

6.1.3 Handling Trials and Conditions

TrialHandler

This is what underlies the random and sequential loop types in *Builder*, they work using the *method of constants*. The `TrialHandler` presents a predetermined list of conditions in either a sequential or random (without replacement) order.

see *TrialHandler* for more details.

TrialHandlerExt (For oddball paradigms)

For handling trial sequences in a *non-counterbalanced design* (i.e. *oddball paradigms*, https://en.wikipedia.org/wiki/Oddball_paradigm). The oddball paradigm is very popular in EEG research.

Its functions are a superset of the class `TrialHandler`, and as such, can also be used for normal trial handling.

see *TrialHandlerExt* for more details.

StairHandler

This generates the next trial using an *adaptive staircase*. The conditions are not predetermined and are generated based on the participant's responses.

Staircases are predominately used in psychophysics to measure the discrimination and detection thresholds. However they can be used in any experiment which varies a numeric value as a result of a 2 alternative forced choice (2AFC) response.

The `StairHandler` systematically generates numbers based on staircase parameters. These can then be used to define a stimulus parameter e.g. spatial frequency, stimulus presentation duration. If the participant gives the incorrect response the number generated will get larger and if the participant gives the correct response the number will get smaller.

see *StairHandler* for more details

6.1.4 Global Event Keys

Global event keys are single keys (or combinations of a single key and one or more “modifier” keys such as Ctrl, Alt, etc.) with an associated Python callback function. This function will be executed if the key (or key/modifiers combination) was pressed.

Note: Global event keys only work with the *pyglet* backend, which is the default.

fully automatically monitors and processes key presses during most portions of the experimental run, for example during *core.wait()* periods, or when calling *win.flip()*. If a global event key press is detected, the specified function will be run immediately. You are not required to manually poll and check for key presses. This can be particularly useful to implement a global “shutdown” key, or to trigger laboratory equipment on a key press when testing your experimental script – without cluttering the code. But of course the application is not limited to these two scenarios. In fact, you can associate any Python function with a global event key.

All active global event keys are stored in *event.globalKeys*.


```
>>> args = (1, 2)
>>> kwargs = dict(foo=3, bar=4)
>>> name = 'my name'
```

Note: Even when intending to pass only a single positional argument, *args* must be a list or tuple, e.g., *args = [1]* or *args = (1)*.

Finally, specify the key and a combination of modifiers. While key names are just strings, modifiers are lists or tuples of modifier names.

```
>>> key = 'b'
>>> modifiers = ['ctrl', 'alt']
```

Note: Even when specifying only a single modifier key, *modifiers* must be a list or tuple, e.g., *modifiers = ['ctrl']* or *modifiers = ('ctrl')*.

We are now ready to create the global event key.

```
>>> event.globalKeys.add(key=key, modifiers=modifiers,
... func=myfunc2, func_args=args, func_kwargs=kwargs,
... name=name)
```

Check that the global event key was successfully added.

```
>>> event.globalKeys
<_GlobalEventKeys :
  [A] -> 'myfunc' <function myfunc at 0x10669ba28>
  [CTRL] + [ALT] + [B] -> 'my name' <function myfunc2 at 0x112eecb90>
>
```

The key combination *[CTRL] + [ALT] + [B]* is now associated with the function *myfunc2*, which will be called in the following way:

```
myfunc2(1, 2, foo=2, bar=4)
```

Indexing

event.globalKeys can be accessed like an ordinary dictionary. The index keys are *(key, modifiers)* namedtuples.

```
>>> event.globalKeys.keys()
[_IndexKey(key='a', modifiers=()), _IndexKey(key='b', modifiers=('ctrl', 'alt'))]
```

To access the global event associated with the key combination *[CTRL] + [ALT] + [B]*, we can do

```
>>> event.globalKeys['b', ['ctrl', 'alt']]
_GlobalEvent(func=<function myfunc2 at 0x112eecb90>, func_args=(1, 2), func_kwargs={
↳ 'foo': 3, 'bar': 4}, name='my name')
```

To make access more convenient, specifying the modifiers is optional in case none were passed to `psychoPy.event.globalKeys.add()` when the global event key was added, meaning the following commands are identical.

```
>>> event.globalKeys['a', ()]
_GlobalEvent(func=<function myfunc at 0x10669ba28>, func_args=(), func_kwargs={},
↳name='myfunc')
>>> event.globalKeys['a']
_GlobalEvent(func=<function myfunc at 0x10669ba28>, func_args=(), func_kwargs={},
↳name='myfunc')
```

All elements of a global event can be accessed directly.

```
>>> index = ('b', ['ctrl', 'alt'])
>>> event.globalKeys[index].func
<function myfunc2 at 0x112eecb90>
>>> event.globalKeys[index].func_args
(1, 2)
>>> event.globalKeys[index].func_kwargs
{'foo': 3, 'bar': 4}
>>> event.globalKeys[index].name
'my name'
```

Number of active event keys

The number of currently active event keys can be retrieved by passing `event.globalKeys` to the `len()` function.

```
>>> len(event.globalKeys)
2
```

Removing global event keys

There are three ways to remove global event keys:

- using `psychoPy.event.globalKeys.remove()`,
- using `del`, and
- using `psychoPy.event.globalKeys.pop()`.

`psychoPy.event.globalKeys.remove()`

To remove a single key, pass the key name and modifiers (if any) to `psychoPy.event.globalKeys.remove()`.

```
>>> event.globalKeys.remove(key='a')
```

A convenience method to quickly delete *all* global event keys is to pass `key='all'`

```
>>> event.globalKeys.remove(key='all')
```

del

Like with other dictionaries, items can be removed from `event.globalKeys` by using the `del` statement. The provided index key must be specified as described in [Indexing](#).

```
>>> index = ('b', ['ctrl', 'alt'])
>>> del event.globalKeys[index]
```

psychoPy.event.globalKeys.pop()

Again, as other dictionaries, `event.globalKeys` provides a `pop` method to retrieve an item and remove it from the dict. The first argument to `pop` is the index key, specified as described in [Indexing](#). The second argument is optional. Its value will be returned in case no item with the matching indexing key could be found, for example if the item had already been removed previously.

```
>>> r = event.globalKeys.pop('a', None)
>>> print(r)
_GlobalEvent(func=<function myfunc at 0x10669ba28>, func_args=(), func_kwargs={},
↳name='myfunc')
>>> r = event.globalKeys.pop('a', None)
>>> print(r)
None
```

Global shutdown key

The preferences for `shutdownKey` and `shutdownKeyModifiers` (both unset by default) will be used to automatically create a global shutdown key. To demonstrate this automated behavior, let us first change the preferences programmatically (these changes will be lost when quitting the current Python session).

```
>>> from psychoPy.preferences import prefs
>>> prefs.general['shutdownKey'] = 'q'
```

We can now check if a global shutdown key has been automatically created.

```
>>> from psychoPy import event
>>> event.globalKeys
<_GlobalEventKeys :
  [Q] -> 'shutdown (auto-created from prefs)' <function quit at 0x10c171938>
>
```

And indeed, it worked!

What happened behind the scenes? When importing the `psychoPy.event` module, the initialization of `event.globalKeys` checked for valid shutdown key preferences and automatically initialized a shutdown key accordingly. This key is associated with the `:func:~`psychoPy.core.quit`` function, which will shut down.

```
>>> from psychoPy.core import quit
>>> event.globalKeys['q'].func == quit
True
```

Of course you can very easily add a global shutdown key manually, too. You simply have to associate a key with `:func:~`psychoPy.core.quit``.

```
>>> from psychopy import core, event
>>> event.globalKeys.add(key='q', func=core.quit, name='shutdown')
```

That's it!

A working example

In the above code snippets, our global event keys were not actually functional, as we didn't create a window, which is required to actually collect the key presses. Our working example will thus first create a window and then add global event keys to change the window color and quit the experiment, respectively.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import print_function
from psychopy import core, event, visual

def change_color(win, log=False):
    win.color = 'blue' if win.color == 'gray' else 'gray'
    if log:
        print('Changed color to %s' % win.color)

win = visual.Window(color='gray')
text = visual.TextStim(win,
                       text='Press C to change color,\n CTRL + Q to quit.')

# Global event key to change window background color.
event.globalKeys.add(key='c',
                    func=change_color,
                    func_args=[win],
                    func_kwargs=dict(log=True),
                    name='change window color')

# Global event key (with modifier) to quit the experiment ("shutdown key").
event.globalKeys.add(key='q', modifiers=['ctrl'], func=core.quit)

while True:
    text.draw()
    win.flip()
```

6.2 Tutorials

6.2.1 Tutorial 1: Generating your first stimulus

A tutorial to get you going with your first stimulus display.

Know your monitor

has been designed to handle your screen calibrations for you. It is also designed to operate (if possible) in the final experimental units that you like to use e.g. degrees of visual angle.

In order to do this needs to know a little about your monitor. There is a GUI to help with this (select `MonitorCenter` from the tools menu of `PsychoPyIDE` or run `...site-packages/monitors/MonitorCenter.py`).

In the `MonitorCenter` window you can create a new monitor name, insert values that describe your monitor and run calibrations like gamma corrections. For now you can just stick to the `[testMonitor]` but give it correct values for your screen size in number of pixels and width in cm.

Now, when you create a window on your monitor you can give it the name 'testMonitor' and stimuli will know how they should be scaled appropriately.

Your first stimulus

Building stimuli is extremely easy. All you need to do is create a `Window`, then some stimuli. Draw those stimuli, then update the window. has various other useful commands to help with timing too. Here's an example. Type it into a coder window, save it somewhere and press run.

```

1 from psychopy import visual, core # import some libraries from PsychoPy
2
3 #create a window
4 mywin = visual.Window([800,600], monitor="testMonitor", units="deg")
5
6 #create some stimuli
7 grating = visual.GratingStim(win=mywin, mask="circle", size=3, pos=[-4,0], sf=3)
8 fixation = visual.GratingStim(win=mywin, size=0.5, pos=[0,0], sf=0, rgb=-1)
9
10 #draw the stimuli and update the window
11 grating.draw()
12 fixation.draw()
13 mywin.update()
14
15 #pause, so you get a chance to see it!
16 core.wait(5.0)

```

Note: For those new to Python. Did you notice that the grating and the fixation stimuli both call `GratingStim` but have different arguments? One of the nice features about python is that you can select which arguments to set. `GratingStim` has over 15 arguments that can be set, but the others just take on default values if they aren't needed.

That's a bit easy though. Let's make the stimulus move, at least! To do that we need to create a loop where we change the phase (or orientation, or position...) of the stimulus and then redraw. Add this code in place of the drawing code above:

```

for frameN in range(200):
    grating.setPhase(0.05, '+') # advance phase by 0.05 of a cycle
    grating.draw()
    fixation.draw()
    mywin.update()

```

That ran for 200 frames (and then waited 5 seconds as well). Maybe it would be nicer to keep updating until the user hits a key instead. That's easy to add too. In the first line add `event` to the list of modules you'll import. Then replace the line:

```
for frameN in range(200):
```

with the line:

```
while True: #this creates a never-ending loop
```

Then, within the loop (make sure it has the same indentation as the other lines) add the lines:

```
    if len(event.getKeys())>0:
        break
    event.clearEvents()
```

the first line counts how many keys have been pressed since the last frame. If more than zero are found then we break out of the never-ending loop. The second line clears the event buffer and should always be called after you've collected the events you want (otherwise it gets full of events that we don't care about like the mouse moving around etc...).

Your finished script should look something like this:

```
1 from psychopy import visual, core, event #import some libraries from PsychoPy
2
3 #create a window
4 mywin = visual.Window([800,600],monitor="testMonitor", units="deg")
5
6 #create some stimuli
7 grating = visual.GratingStim(win=mywin, mask='circle', size=3, pos=[-4,0], sf=3)
8 fixation = visual.GratingStim(win=mywin, size=0.2, pos=[0,0], sf=0, rgb=-1)
9
10 #draw the stimuli and update the window
11 while True: #this creates a never-ending loop
12     grating.setPhase(0.05, '+') #advance phase by 0.05 of a cycle
13     grating.draw()
14     fixation.draw()
15     mywin.flip()
16
17     if len(event.getKeys())>0:
18         break
19     event.clearEvents()
20
21 #cleanup
22 mywin.close()
23 core.quit()
```

There are several more simple scripts like this in the demos menu of the Coder and Builder views and many more to download. If you're feeling like something bigger then go to *Tutorial 2: Measuring a JND using a staircase procedure* which will show you how to build an actual experiment.

6.2.2 Tutorial 2: Measuring a JND using a staircase procedure

This tutorial builds an experiment to test your just-noticeable-difference (JND) to orientation, that is it determines the smallest angular deviation that is needed for you to detect that a gabor stimulus isn't vertical (or at some other reference orientation). The method presents a pair of stimuli at once with the observer having to report with a key press whether the left or the right stimulus was at the reference orientation (e.g. vertical).

You can download the [full code here](#). Note that the entire experiment is constructed of less than 100 lines of code, including the initial presentation of a dialogue for parameters, generation and presentation of stimuli, running the trials, saving data and outputting a simple summary analysis for feedback. Not bad, eh?

There are a great many modifications that can be made to this code, however this example is designed to demonstrate how much can be achieved with very simple code. Modifying existing is an excellent way to begin writing your own scripts, for example you may want to try changing the appearance of the text or the stimuli.

Get info from the user

The first lines of code import the necessary libraries. We need lots of the modules for a full experiment, as well as *numpy* (which handles various numerical/mathematical functions):

```

1  """measure your JND in orientation using a staircase method"""
2  from psychopy import core, visual, gui, data, event
3  from psychopy.tools.filetools import fromFile, toFile
4  import numpy, random

```

Also note that there are two ways to insert comments in Python (and you should do this often!). Using triple quotes, as in `"""Here's my comment"""`, allows you to write a comment that can span several lines. Often you need that at the start of your script to leave yourself a note about the implementation and history of what you've written. For single-line comments, as you'll see below, you can use a simple `#` to indicate that the rest of the line is a comment.

The `try:...except:...` lines allow us to try and load a parameter file from a previous run of the experiment. If that fails (e.g. because the experiment has never been run) then create a default set of parameters. These are easy to store in a python dictionary that we'll call *expInfo*:

```

6  try: # try to get a previous parameters file
7      expInfo = fromFile('lastParams.pickle')
8  except: # if not there then use a default set
9      expInfo = {'observer':'jwp', 'refOrientation':0}
10 expInfo['dateStr'] = data.getDateStr() # add the current time

```

The last line adds the current date to to the information, whether we loaded from a previous run or created default values.

So having loaded those parameters, let's allow the user to change them in a dialogue box (which we'll call `dlg`). This is the simplest form of dialogue, created directly from the dictionary above. the dialogue will be presented immediately to the user and the script will wait until they hit *OK* or *Cancel*.

If they hit *OK* then `dlg.OK=True`, in which case we'll use the updated values and save them straight to a parameters file (the one we try to load above).

If they hit *Cancel* then we'll simply quit the script and not save the values.

```

11 # present a dialogue to change params
12 dlg = gui.DlgFromDict(expInfo, title='simple JND Exp', fixed=['dateStr'])
13 if dlg.OK:
14     toFile('lastParams.pickle', expInfo) # save params to file for next time
15 else:
16     core.quit() # the user hit cancel so exit

```

Setup the information for trials

We'll create a file to which we can output some data as text during each trial (as well as *outputting a binary file* at the end of the experiment). actually has supporting functions to do this automatically, but here we're showing you the manual way to do it.

We'll create a filename from the subject+date+”.csv” (note how easy it is to concatenate strings in python just by ‘adding’ them). *csv* files can be opened in most spreadsheet packages. Having opened a text file for writing, the last line shows how easy it is to send text to this target document.

```

18 # make a text file to save data
19 fileName = expInfo['observer'] + expInfo['dateStr']
20 dataFile = open(fileName+'.csv', 'w') # a simple text file with 'comma-separated-
    ↳values'
21 dataFile.write('targetSide,oriIncrement,correct\n')
```

allows us to set up an object to handle the presentation of stimuli in a staircase procedure, the *StairHandler*. This will define the increment of the orientation (i.e. how far it is from the reference orientation). The staircase can be configured in many ways, but we'll set it up to begin with an increment of 20deg (very detectable) and home in on the 80% threshold value. We'll step up our increment every time the subject gets a wrong answer and step down if they get three right answers in a row. The step size will also decrease after every 2 reversals, starting with an 8dB step (large) and going down to 1dB steps (smallish). We'll finish after 50 trials.

```

23 # create the staircase handler
24 staircase = data.StairHandler(startVal = 20.0,
25                               stepType = 'db', stepSizes=[8,4,4,2],
26                               nUp=1, nDown=3, # will home in on the 80% threshold
27                               nTrials=1)
```

Build your stimuli

Now we need to create a window, some stimuli and timers. We need a *~psychopy.visual.Window* in which to draw our stimuli, a fixation point and two *~psychopy.visual.GratingStim* stimuli (one for the target probe and one as the foil). We can have as many timers as we like and reset them at any time during the experiment, but I generally use one to measure the time since the experiment started and another that I reset at the beginning of each trial.

```

29 # create window and stimuli
30 win = visual.Window([800,600],allowGUI=True,
31                     monitor='testMonitor', units='deg')
32 foil = visual.GratingStim(win, sf=1, size=4, mask='gauss',
33                            ori=expInfo['refOrientation'])
34 target = visual.GratingStim(win, sf=1, size=4, mask='gauss',
35                              ori=expInfo['refOrientation'])
36 fixation = visual.GratingStim(win, color=-1, colorSpace='rgb',
37                                tex=None, mask='circle', size=0.2)
38 # and some handy clocks to keep track of time
39 globalClock = core.Clock()
40 trialClock = core.Clock()
```

Once the stimuli are created we should give the subject a message asking if they're ready. The next two lines create a pair of messages, then draw them into the screen and then update the screen to show what we've drawn. Finally we issue the command `event.waitKeys()` which will wait for a keypress before continuing.

```

42 # display instructions and wait
43 message1 = visual.TextStim(win, pos=[0,+3],text='Hit a key when ready.')
```

(continues on next page)

(continued from previous page)

```

44 message2 = visual.TextStim(win, pos=[0,-3],
45     text="Then press left or right to identify the %.1f deg probe." %expInfo[
    ↪ 'refOrientation'])
46 message1.draw()
47 message2.draw()
48 fixation.draw()
49 win.flip() #to show our newly drawn 'stimuli'
50 #pause until there's a keypress
51 event.waitKeys()

```

Control the presentation of the stimuli

OK, so we have everything that we need to run the experiment. The following uses a for-loop that will iterate over trials in the experiment. With each pass through the loop the `staircase` object will provide the new value for the intensity (which we will call `thisIncrement`). We will randomly choose a side to present the target stimulus using `numpy.random.random()`, setting the position of the target to be there and the foil to be on the other side of the fixation point.

```

53 for thisIncrement in staircase: # will continue the staircase until it terminates!
54     # set location of stimuli
55     targetSide= random.choice([-1,1]) # will be either +1(right) or -1(left)
56     foil.setPos([-5*targetSide, 0])
57     target.setPos([5*targetSide, 0]) # in other location

```

Then set the orientation of the foil to be the reference orientation plus `thisIncrement`, draw all the stimuli (including the fixation point) and update the window.

```

59     # set orientation of probe
60     foil.setOri(expInfo['refOrientation'] + thisIncrement)
61
62     # draw all stimuli
63     foil.draw()
64     target.draw()
65     fixation.draw()
66     win.flip()

```

Wait for presentation time of 500ms and then blank the screen (by updating the screen after drawing just the fixation point).

```

68     # wait 500ms; but use a loop of x frames for more accurate timing
69     core.wait(0.5)

```

(This is not the most precise way to time your stimuli - you'll probably overshoot by one frame - but its easy to understand. allows you to present a stimulus for acertain number of screen refreshes instead which is better for short stimuli.)

Get input from the subject

Still within the for-loop (note the level of indentation is the same) we need to get the response from the subject. The method works by starting off assuming that there hasn't yet been a response and then waiting for a key press. For each key pressed we check if the answer was correct or incorrect and assign the response appropriately, which ends the trial. We always have to clear the event buffer if we're checking for key presses like this

```

75     # get response
76     thisResp=None
77     while thisResp==None:
78         allKeys=event.waitKeys()
79         for thisKey in allKeys:
80             if thisKey=='left':
81                 if targetSide==-1: thisResp = 1 # correct
82                 else: thisResp = -1           # incorrect
83             elif thisKey=='right':
84                 if targetSide== 1: thisResp = 1 # correct
85                 else: thisResp = -1           # incorrect
86             elif thisKey in ['q', 'escape']:
87                 core.quit() # abort experiment
88             event.clearEvents() # clear other (eg mouse) events - they clog the buffer
    
```

Now we must tell the staircase the result of this trial with its `addData()` method. Then it can work out whether the next trial is an increment or decrement. Also, on each trial (so still within the for-loop) we may as well save the data as a line of text in that .csv file we created earlier.

```

90     # add the data to the staircase so it can calculate the next level
91     staircase.addData(thisResp)
92     dataFile.write('%i,%.3f,%i\n' %(targetSide, thisIncrement, thisResp))
93     core.wait(1)
    
```

Output your data and clean up

OK! We're basically done! We've reached the end of the for-loop (which occurred because the staircase terminated) which means the trials are over. The next step is to close the text data file and also save the staircase as a binary file (by 'pickling' the file in Python speak) which maintains a lot more info than we were saving in the text file.

```

95     # staircase has ended
96     dataFile.close()
97     staircase.saveAsPickle(fileName) # special python binary file to save all the info
    
```

While we're here, it's quite nice to give some immediate feedback to the user. Let's tell them the intensity values at the all the reversals and give them the mean of the last 6. This is an easy way to get an estimate of the threshold, but we might be able to do a better job by trying to reconstruct the psychometric function. To give that a try see the staircase analysis script of [Tutorial 3](#).

Having saved the data you can give your participant some feedback and quit!

```

99     # give some output to user in the command line in the output window
100    print('reversals:')
101    print(staircase.reversalIntensities)
102    approxThreshold = numpy.average(staircase.reversalIntensities[-6:])
103    print('mean of final 6 reversals = %.3f' % (approxThreshold))
104
105    # give some on-screen feedback
106    feedback1 = visual.TextStim(
    
```

(continues on next page)

(continued from previous page)

```

107     win, pos=[0,+3],
108     text='mean of final 6 reversals = %.3f' % (approxThreshold))
109
110 feedback1.draw()
111 fixation.draw()
112 win.flip()
113 event.waitKeys() # wait for participant to respond
114
115 win.close()
116 core.quit()

```

6.2.3 Tutorial 3: Analysing data in Python

You could simply output your data as tab- or comma-separated text files and analyse the data in some spreadsheet package. But the `matplotlib` library in Python also allows for very neat and simple creation of publication-quality plots.

This script shows you how to use a couple of functions from to open some data files (`psychopy.gui.fileOpenDlg()`) and create a psychometric function out of some staircase data (`psychopy.data.functionFromStaircase()`).

`Matplotlib` is then used to plot the data.

Note: `Matplotlib` and `pylab`. `Matplotlib` is a python library that has similar command syntax to most of the plotting functions in `Matlab(tm)`. It can be imported in different ways; the `import pylab` line at the beginning of the script is the way to import `matplotlib` as well as a variety of other scientific tools (that aren't strictly to do with plotting *per se*).

```

1  #This analysis script takes one or more staircase datafiles as input
2  #from a GUI. It then plots the staircases on top of each other on
3  #the left and a combined psychometric function from the same data
4  #on the right
5
6  from psychopy import data, gui, core
7  from psychopy.tools.filetools import fromFile
8  import pylab
9
10 #Open a dialog box to select files from
11 files = gui.fileOpenDlg('.')
12 if not files:
13     core.quit()
14
15 #get the data from all the files
16 allIntensities, allResponses = [],[]
17 for thisFileName in files:
18     thisDat = fromFile(thisFileName)
19     allIntensities.append( thisDat.intensities )
20     allResponses.append( thisDat.data )
21
22 #plot each staircase
23 pylab.subplot(121)
24 colors = 'brgkcmbrgkcm'
25 lines, names = [],[]

```

(continues on next page)

(continued from previous page)

```
26 for fileN, thisStair in enumerate(allIntensities):
27     #lines.extend(pylab.plot(thisStair))
28     #names = files[fileN]
29     pylab.plot(thisStair, label=files[fileN])
30     #pylab.legend()
31
32     #get combined data
33     combinedInten, combinedResp, combinedN = \
34         data.functionFromStaircase(allIntensities, allResponses, 5)
35     #fit curve - in this case using a Weibull function
36     fit = data.FitWeibull(combinedInten, combinedResp, guess=[0.2, 0.5])
37     smoothInt = pylab.arange(min(combinedInten), max(combinedInten), 0.001)
38     smoothResp = fit.eval(smoothInt)
39     thresh = fit.inverse(0.8)
40     print(thresh)
41
42     #plot curve
43     pylab.subplot(122)
44     pylab.plot(smoothInt, smoothResp, '-')
45     pylab.plot([thresh, thresh],[0,0.8], '--'); pylab.plot([0, thresh],\
46     [0.8,0.8], '--')
47     pylab.title('threshold = %0.3f' %(thresh))
48     #plot points
49     pylab.plot(combinedInten, combinedResp, 'o')
50     pylab.ylim([0,1])
51
52     pylab.show()
```


RUNNING AND SHARING STUDIES ONLINE

Online studies are realized via [PsychoJS](#); the online counterpart of `PsychoPy`. To run your study online, these are the basic steps:

- Check the features supported by PsychoJS to ensure the components you need will work online.
- Make your experiment in *Builder*.
- Configure the online settings of your experiment.
- Launch your study on Pavlovia.org.

When making an experiment to run online, there are a few important considerations to make and we **highly** recommend reading through the considerations below, as they could save a lot of time in the long run!

- Using resources in online studies.
- Multisession testing, Counterbalancing, and multiplayer games
- Caveats and cautions (timing accuracy and web-browser support).

7.1 Related links

7.1.1 Troubleshooting Online Studies

Sometimes experiments might work perfectly locally, when created and run in the PsychoPy application, but the same experiment might not behave as you expect when you try to run them online, through pavlovia.org. While this page cannot hope to address all of the possible issues you may encounter, it should help you understand the different types of errors and help you give more detailed information if you ask for support on the PsychoPy forums.

Getting Started

PsychoPy Builder is your friend

1. Check whether the features you are using are supported online via our [onlineStatus](#) page.
2. Don't try to code in PsychoJS directly.
3. Don't try to edit JavaScript files on Pavlovia directly. Make changes via Builder.
4. Each Builder (psyexp) file should be in its own dedicated local folder, which should not be in an area currently under version control (e.g. a github project, Google drive or Onedrive). This folder should only contain subfolders that pertain to the experiment.
5. Upload your files to Gitlab by synchronising using PsychoPy Builder, rather than using Git commands.

6. Code components should be set to Auto translate (“Code Type” > Auto > JS) unless you know why you need to use different code for Python and JavaScript.
7. Code components should normally be moved to the top of their respective routines. Your code is executed in order from left to right (in the flow) and from top to bottom (within each routine).
8. Experiment Settings / Online / Output path should be blank.
9. Resources (spreadsheets, images, etc.) should be in the same folder as the psyexp file or a sub-folder. Resources that are selected via code components should be added via Experiment Settings / Online / Additional Resources (see how to configureOnline) or a Resource Manager Component. See handlingOnlineResources for more information.

Running the latest version of your experiment

When you synchronise changes to your experiment, you may need to clear your browser cache to see those changes online (using Ctrl-F5, Ctrl-Shift-R or equivalent). If this does not work use an incognito browser tab. A participant will not need to do this, so long as they have not already tried a previous version of your experiment.

Developer Tools

Use Developer Tools (Ctl-Shift-I in Windows/Chrome, Cmd-Opt-J or Cmd-Opt-I in Mac/Chrome, F12 in IE/Edge, Ctrl-Shift-J in Windows/Safari, Ctrl-Opt-J in Mac/Safari) to view errors via the browser console if you aren’t getting sufficient information from PsychoPy. You can also add `print(X)` (which translates to `console.log(X)`; where X refers to the name of your variable) to check the value of a variable X at a particular point.

Tutorial [tutorial_js_console_log](#)

Types of Errors

Errors in your experiment can manifest in multiple ways. The easiest way to categorise the different types of error message is based on where they appear.

- **Builder Errors**
 - Python syntax errors
 - Builder runtime errors
 - Synchronisation errors
- **Browser Errors**
 - Launch errors
 - Resource errors
 - Semantic errors
- **Unexpected Behaviour**

Python Syntax Errors (seen in Auto-translate code components)

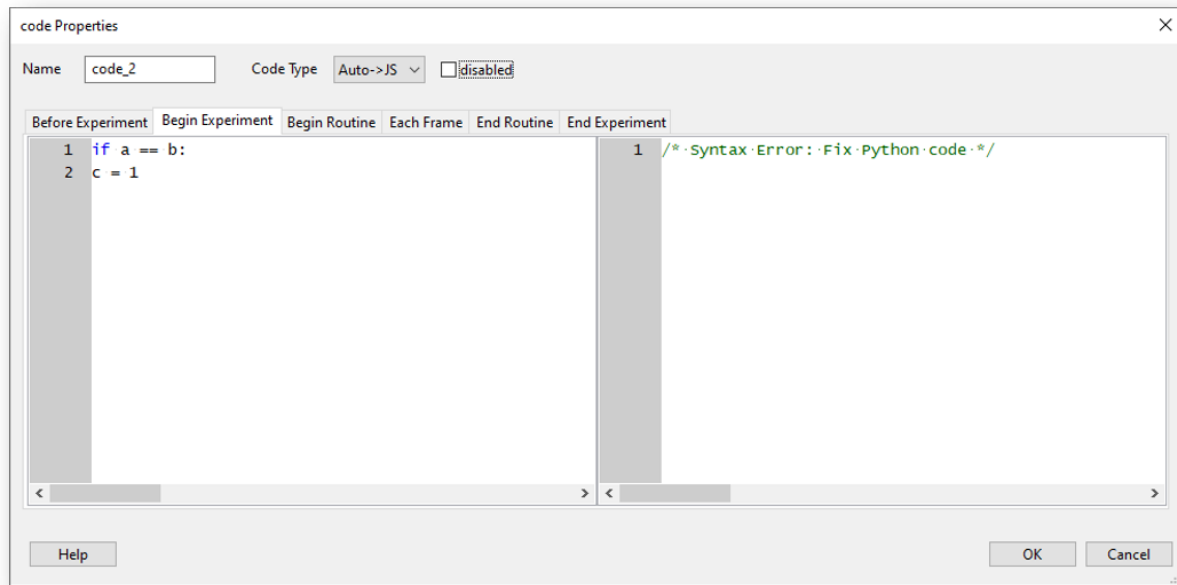


Fig. 7.1: A code component used in PsychoPy Builder. In this example, “Code Type” is set to “Auto > JS” meaning python code (on the left) will transpile to JavaScript (on the right). In this example there is a python coding error, which means the transpilation cannot occur.

One of the advantages of using auto translate code components is that the transpiler is continuously checking your code in order to translate it to JavaScript. If you have a syntax error in your Python code, the JavaScript translation will be `/* Syntax Error: Fix Python code */`. If you get this type of error then your Python code probably won’t run locally, and no translated code will be added to the JavaScript version.

Note: “Old style” string formatting (using a `%` operator) works in Python but gives a syntax error in JavaScript but string interpolation (f-strings) is fine.

Synchronisation Errors (seen in the PsychoPy Runner Stdout)

Errors that occur here during synchronisation are often related to the connection to the gitlab repository on Pavlovia. The Stdout will contain a number of messages. Focus on errors (not warnings) which appear near the top or bottom of the output that has just been generated. If you need to recreate a new project then you may need to delete the local hidden `.git` folder to sever the old connection. If the error message is not related to the git connection, this [flow chart](#) might be helpful.

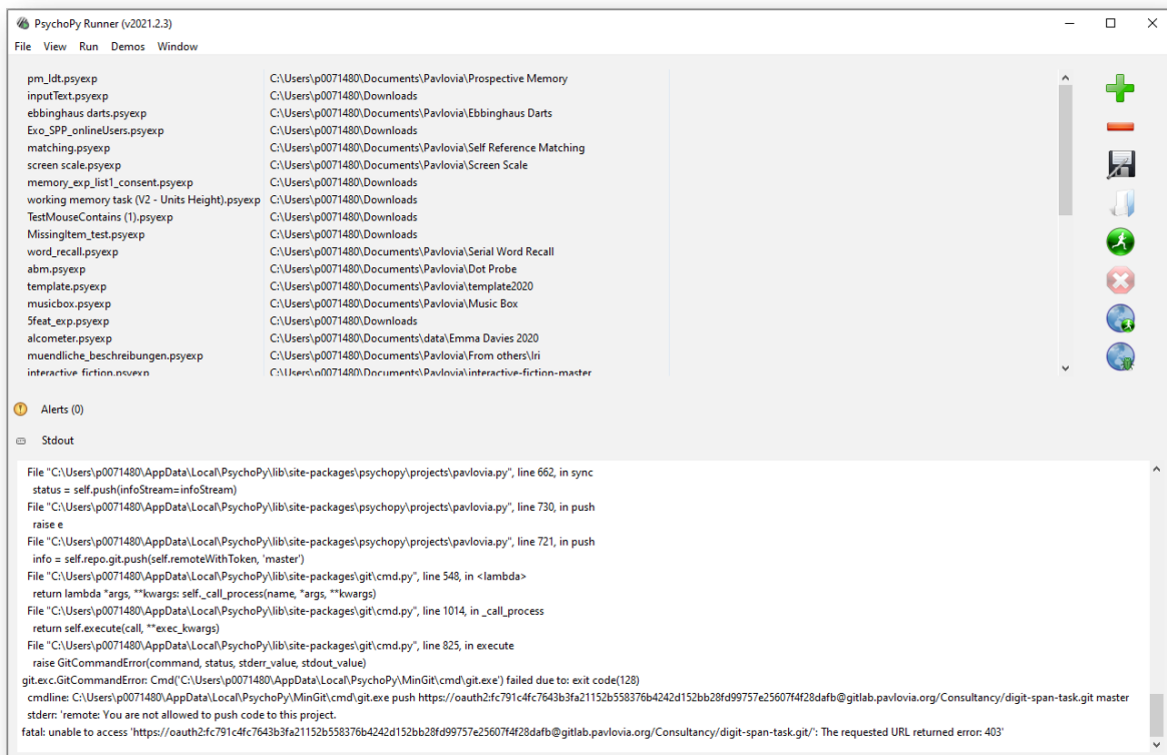


Fig. 7.2: An example “synchronisation error” as shown in PsychoPy Runner. In this example the experimenter is attempting to synchronise an experiment while logged into a different Pavlovia account in PsychoPy Builder.

Synchronisation Errors (seen in a pop-up when synchronising)

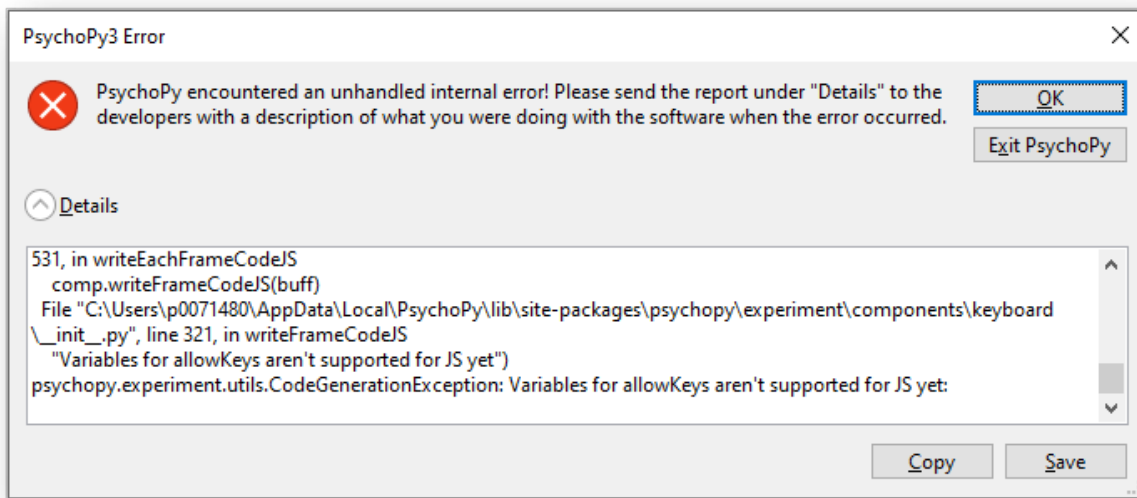


Fig. 7.3: An example “synchronisation error” as shown in PsychoPy Builder. In this example the experimenter has set the *Allowed keys* of a keyboard component as a variable, which is not yet supported in PsychoJS.

Errors occur here when PsychoPy is unable to create a JavaScript file from your Builder file. They are usually related to your custom code components, but can be caused by unexpected parameters in your other components. These errors will prevent your JavaScript files from being created and therefore stop you making any changes to previous versions you may have successfully synchronised. See usingPavlovioa for more information.

Launch Errors (stuck on “initialising the experiment”)

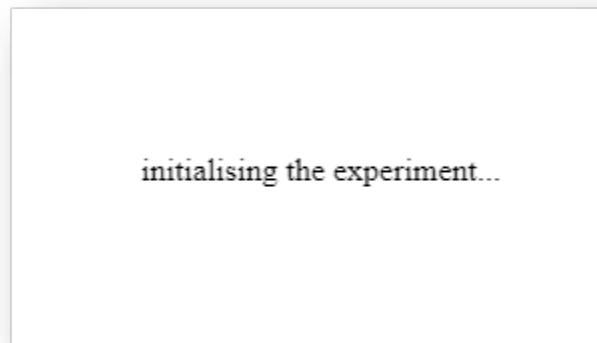


Fig. 7.4: The “initialising the experiment” message shown when launching and experiment in pavlovioa.org.

If, when you try to launch your experiment, it is stuck on “initialising the experiment” then Pavlovioa has encountered a syntax error in your JavaScript file that wasn’t caught by the checks during synchronisation. The most common

cause for this error is that you are trying to import a Python library, such as random or numpy, which don't exist in JavaScript. Use Developer Tools to look for more information.

Tutorial [tutorial_js_syntax_error](#) experiment

Resource Errors

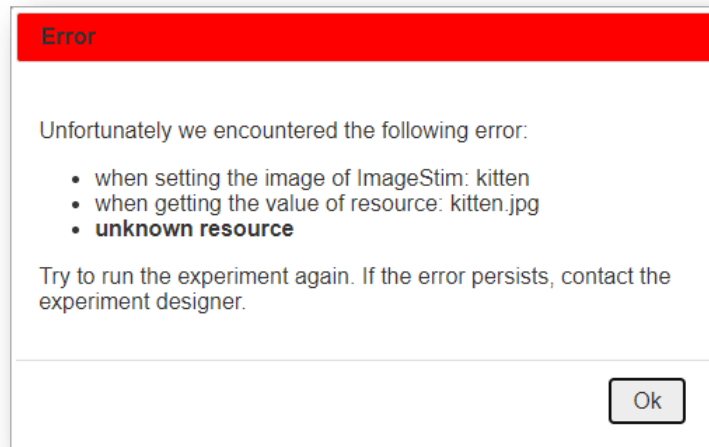


Fig. 7.5: An example “unknown resource” error message as shown in pavlovia.org. In this example the experiment cannot locate an image.

To understand resource errors it is really important to understand handlingOnlineResources - and we recommend you check out this information to understand how to properly load resources in your experiment. This occurs when an additional resource such as a spreadsheet or image file hasn't been made available to the experiment. This can either occur because the file couldn't be found when requested, or because there was an attempt to use the file without downloading it first. These errors are often referred to as network errors, but this does not mean that they are caused by general connectivity issues.

Tutorial [tutorial_js_network_error](#) experiment

Semantic Errors

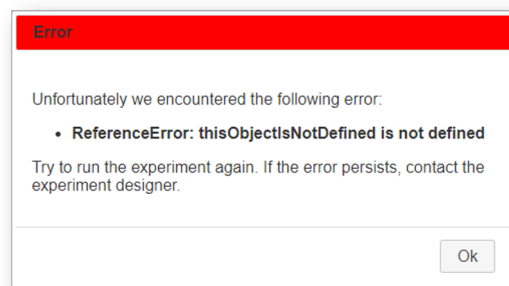


Fig. 7.6: An example “semantic error” where something is not defined (Typically a variable name).

These errors occur when a variable has not been defined or declared in the JavaScript version of your experiment. There are typically two reasons for this error.

1. You may have used a python library of PsychoPy object that does not exist, and is therefore not defined, in JavaScript. For example if you used `np.average([1, 2, 3])` in a code component, you would get the error message “np is not defined” (to avoid this specific error use `average([1, 2, 3])` - dropping the reference to numpy).
2. To define a variable in simply add something like `x = 1` in the Begin Experiment or Begin Routine tab of an auto translate code component.

Most semantic errors can be solved by searching for the text of the error message on the [discourse forum](#). You can also use the Developer Tools to help identify which command is causing the error.

Tutorial [tutorial_js_semantic_error experiment](#)

Unexpected Behaviour

Sometimes your experiment will run without any error messages but something will be missing or wrong. This can occur if:

1. you try to use a component that doesn't yet work online
2. you have code components set to Python only.
3. you use a python function that might work subtly differently in python and JavaScript (for example `pop(0)` will remove the first thing from a list in python, but the last thing from a list in Javascript).

If you're using code components, it's useful to think about the positions of your code components and how they are executed relative to your other components. Since **Begin Routine** code tabs are executed at the same time as **set every repeat** component parameters in top to bottom order. Did you set the parameter before or after it was used? If you something to change during a routine, it needs to be in an **Each Frame** code tab or a **set every frame** component parameter.

Getting Help

Once you have identified the error message or behaviour you are trying to fix, search the [PsychoPy forum](#) for other threads discussing the same issue, using keywords from your error message or issue. Some threads are marked with a tick before the name to indicate that they contain a solution. You may also find the solution in Wakefield Morys-Carter's [PsychoPy to JS crib sheet](#).

If your issue is solved thanks to a solution you found in a thread, we recommend adding a +1 or like reaction to the post that helped you (remember many of those who support our forum are volunteers! so it's useful to show appreciation and indicate to others seeking help which answer was used by others). If a post you create is solved by a suggestion please mark that response with as the “solution”.

If you are unable to solve the problem with existing solutions already posted on the forum then either add a post to a thread which refers to the same issue and doesn't have a solution or start a new thread and include a link to the solution you tried or the most similar thread you have come across in your search.

Creating a New Topic on the forum

Select an appropriate *category*:

- **Online experiments** if you are planning to run your experiment online.
- **Builder** if you are using PsychoPy Builder for a local experiment.
- **Coding** if you are using PsychoPy Coder for a local experiment.
- **Other** if you are having issues that aren't related to a particular experiment.

Give your new topic a useful *title* such as the text of the error message and/or a short clear description of what is going wrong.

Include the *version of PsychoPy* you are using and a usable link to your experiment.

If you have a Browser error near the beginning of your experiment, it is helpful to allow people to try it for themselves. Since Pilot tokens expire, the easiest way to allow others to view your experiment is to set it to RUNNING and allocate it a small number of credits. Add a final routine with a text component that doesn't end (possibly unless you press a key such as = which isn't typically used). You should also set your experiment not to save incomplete results using the Dashboard entry for your project so no credits are consumed during testing.

Since most of the JavaScript code is generated automatically, either from Builder components or by Auto translations in code components it is most useful to show screen shots from Builder (the flow and the relevant routine, plus the contents of the component with the issue). If the issue is with an Auto code component, then you should paste the contents of the Python side as preformatted text, as well as showing the screenshot. Only paste JavaScript from Both and JS only code components to clarify that these have been manually edited.

What next?

We will try to give as much support as possible for free in the public space. However if you are still stuck we can offer paid consultancy options to help debug. You can contact our team directly at consultancy@opensciencetools.org. Consultancy is part of our sustainable model for Open Source Tools and allows us to keep creating free and accessible tools (see *Overview* and read more on [Open Science Tools](#)). Our Science team will be happy to help via one-to-one technical support hours or larger consultancy projects. contains a collection of experiments that you can use as a starting point for your own experiment. Below we explain how you can search for experiments in this collection and contribute your own experiment.

7.1.2 Searching for experiments on Pavlovia

You can search for experiments via the website and from within the PsychoPy Builder.

Via the website

From the home page, you can explore your own existing projects, or other users' public projects. To find a project, go to Pavlovia's [Explore page](#) (see [Fig. 7.7](#)).

When exploring studies online, you are presented with a series of thumbnail images for all of the projects on . See [Fig. 7.8](#).

From the "Explore" page, you can filter projects by setting the filter buttons to a) Public or Private, b) Active or Inactive, and c) sort by number of forks, name, date and number of stars. The default sorting method is Stars. You can also search for projects using the search tool using keywords describing your area of interest, e.g., Stroop, or attention.

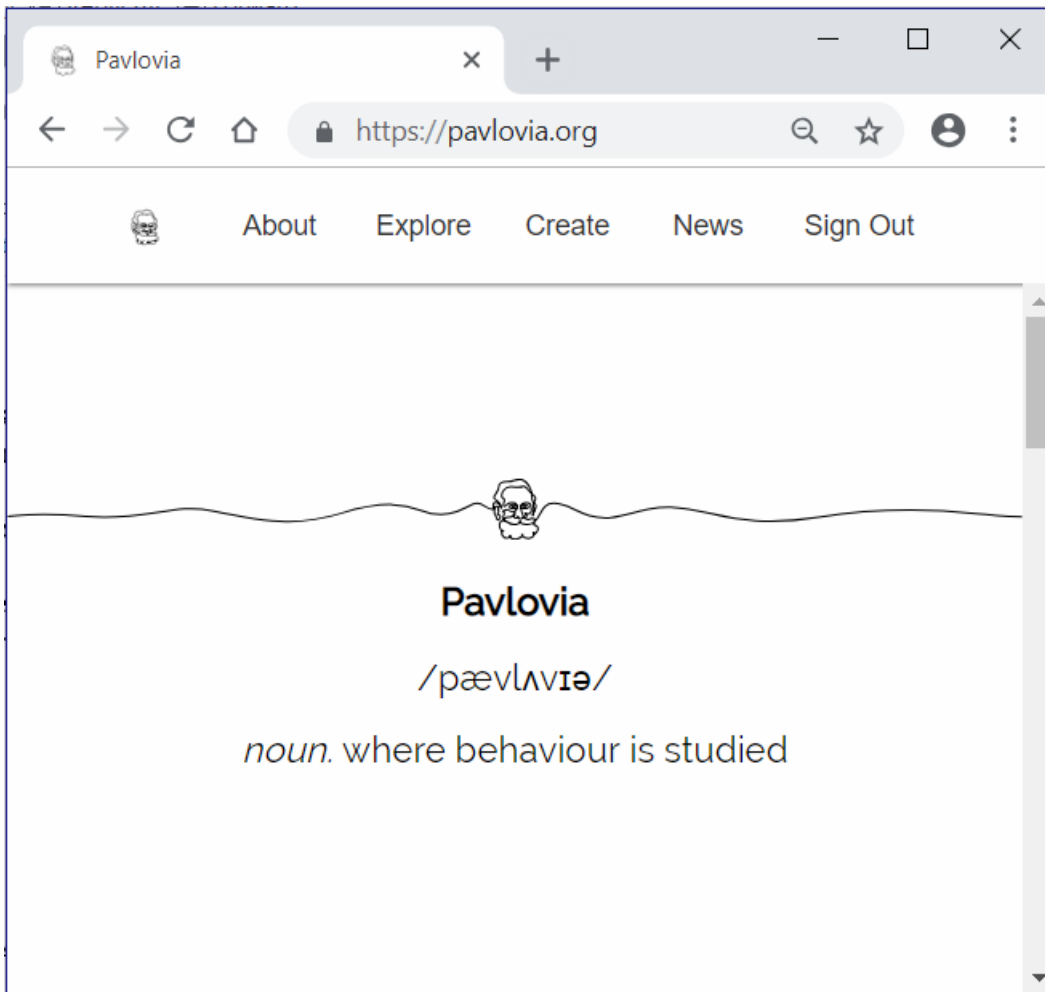


Fig. 7.7: The home page

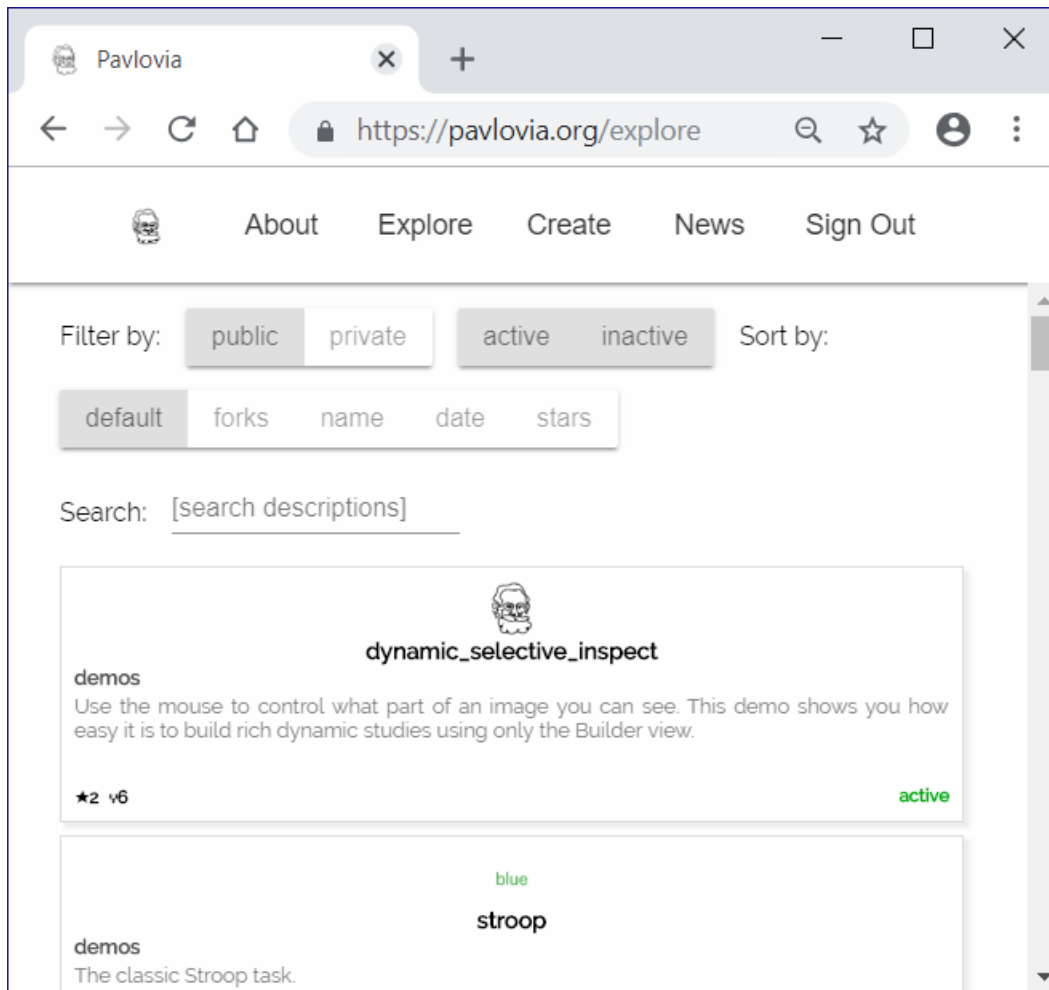


Fig. 7.8: Exploring projects on

Via the PsychoPy Builder

If you wish to search for your own existing projects on , or other users’ public projects, you can also do this via the Builder interface. To search for a project, click button (3) on the Builder Frame in Fig. 7.9.

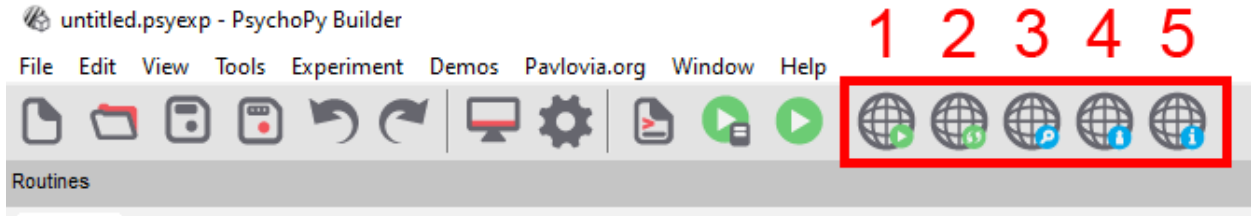


Fig. 7.9: Buttons for running an online study from the PsychoPy Builder.

Following this, a search dialog will appear, see Fig. 7.10. The search dialog presents several options that allow users to search, fork and synchronize projects.

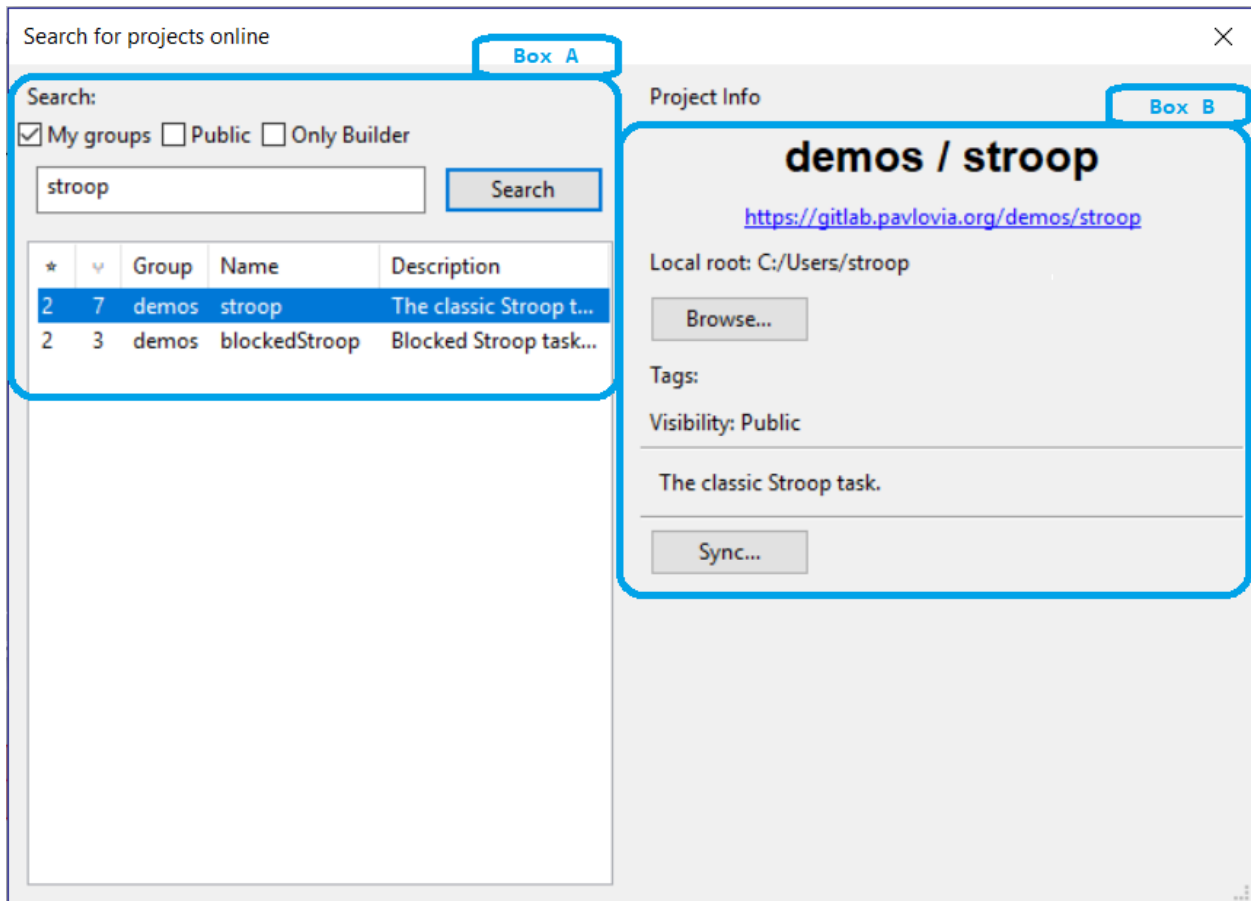


Fig. 7.10: The search dialog in Builder

To search for a project (see Fig. 7.10, Box A), type in search terms in the text box and click the “Search” button to find related projects on . Use the search filters (e.g., “My group”, “Public” etc) above the text box to filter the search output. The output of your search will be listed in the search panel below the search button, where you can select your project of interest.

To **fork and sync a project** is to take your own copy of a project from (*fork*) and copy a version to your local desktop or laptop computer (*sync*). To fork a project, select the local folder to download the project using the “Browse” button, and then click “Sync” when you are ready - (see Fig. 7.10, Box B). You should now have a local copy of the project from ready to run in PsychoPy!

7.1.3 Contributing an experiment to Pavlovia

If you contribute an experiment to Pavlovia, other researchers can access it. Besides contributing to open science, this can be handy if you’ve got an issue with your experiment and would like other researchers to take a look.

Making an experiment public

A public experiment is visible to anyone to clone and fork. To make your experiment public navigate to your experiments’ GitLab page, then select > View code <> > Settings > Permissions (set to public). See Fig. 7.11.

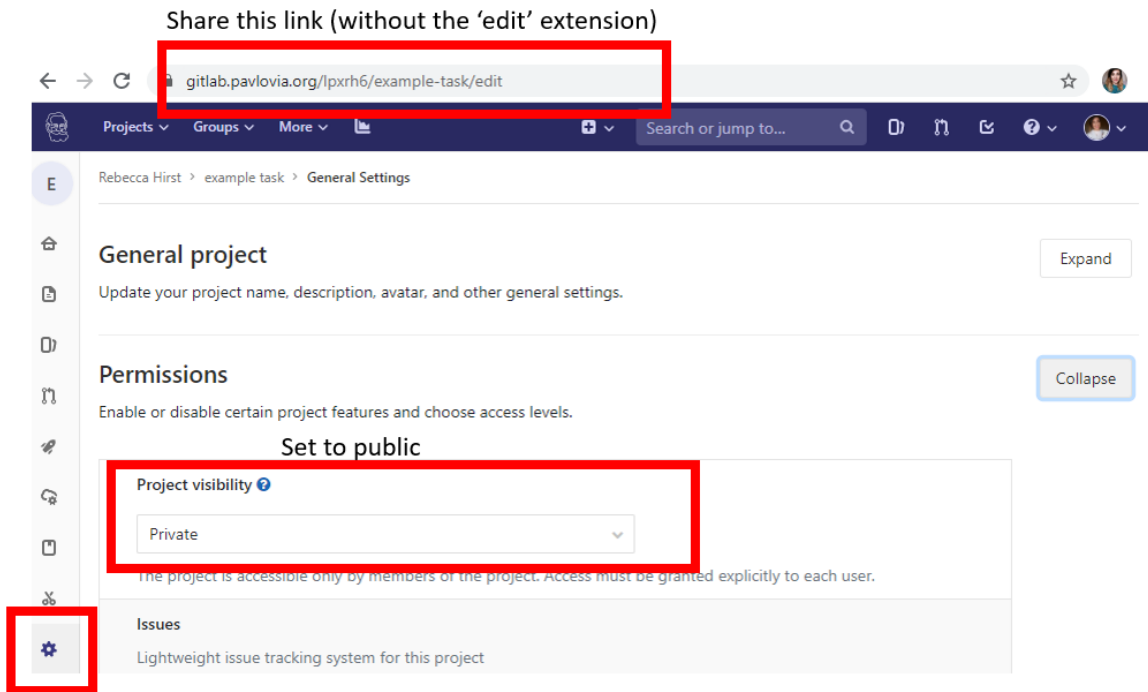


Fig. 7.11: Setting a GitLab project to public access

Adding a team member

If you’d like to share your experiment only with specific researchers, navigate to your experiment, then select > View code <> > settings > Members. At this page: select a member, give them a role (to be able to fork your experiment, they should at least be Developer), optionally an access expiration date, and then add them. See Fig. 7.12.

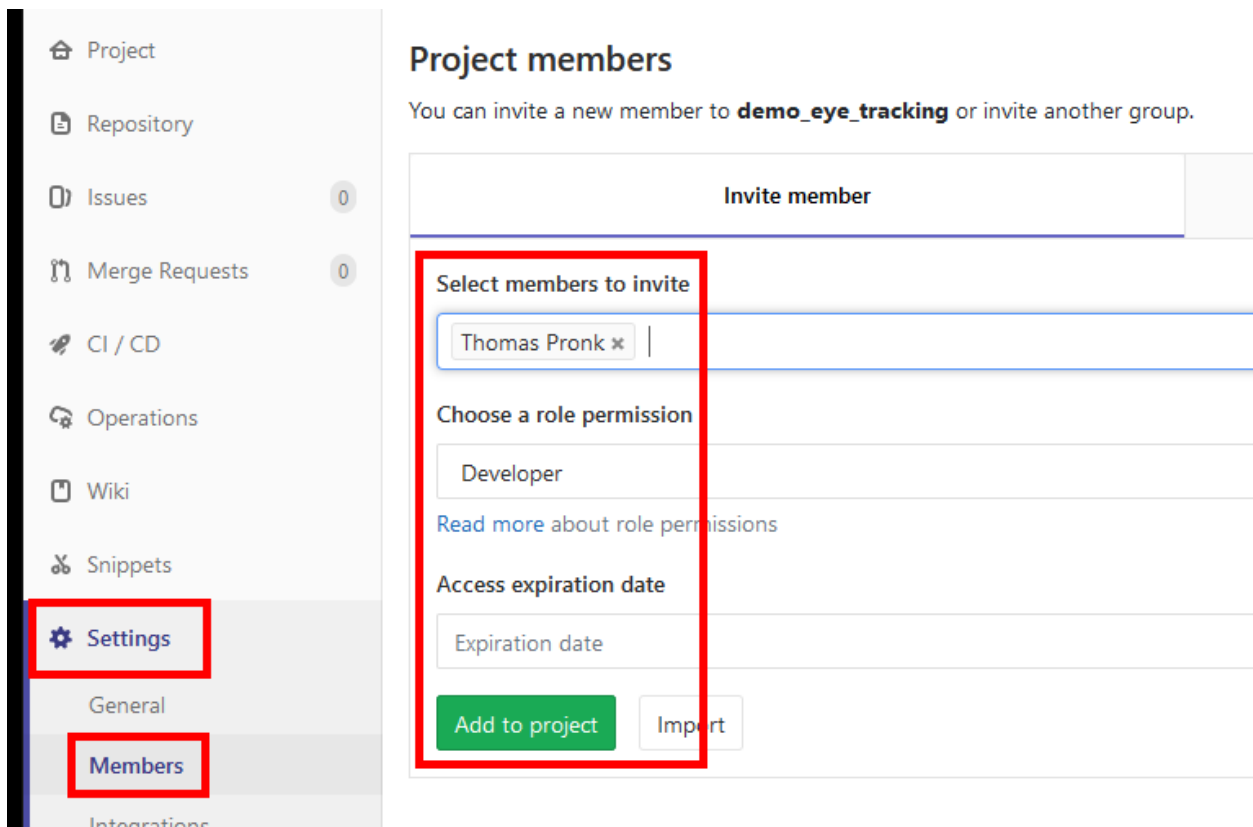



Fig. 7.12: Adding a user to a GitLab project

7.1.4 Recruiting participants and connecting with online services

Having created your study in Builder, uploaded it to Pavlovia, and activated it, you now need to recruit your participants to run the study.

At the simplest level you can get the URL for the study and distribute it to participants manually (e.g. by email or social media). To get the URL to run you can either press the Builder button to “Run online”  and then you can select the URL in the resulting browser window that should appear.

PsychoPy can also connect to a range of other online systems as well, however, some of which are helpful in recruiting participants. Below we describe the general approach before describing the specifics for some common systems:

Recruiting with Sona Systems

Sona Systems is a great platform, particularly for the case where your students are all encouraged to sign up to the platform (e.g. for course credits) and the nrun studies for each other.

Sona Systems have shared an excellent [guide to using Sona with PsychoPy](#) and we would refer you to that for tips to get started.

Recruiting with Prolific

Prolific is a dedicated service designed specifically for behavioural scientists. It aims to provide improved data quality over the likes of Mechanical Turk, with better participant selection and screening, and to provide more ethical pay levels to participants in your study.

As described in the page [Recruiting participants and connecting with online services](#), connecting Prolific to PsychoPy is simply a matter of telling Prolific the URL for your study (including parameters to receive the Study ID etc) and then telling PsychoPy the URL to use when the participant completes the study.

Example link to provide **to Prolific** as your study URL (you will need to replace *myUserName* and *myStudyName*):

```
http://run.pavlovia.org/myUserName/myStudyName/index.html?participant={{%PROLIFIC_PID  
→%}}&study_id={{%STUDY_ID%}}&session={{%SESSION_ID%}}
```

Example link to provide **to PsychoPy** as your completion URL (you will need to change your study ID number):

```
https://app.prolific.co/submissions/complete?cc=T8ZI42EG
```

Further details on how to find and set these links and parameters are as follows. See also [Integrating Prolific with your study](#)

Setting the study URL in Prolific

To recruit participants to your PsychoPy study you should see a screen as in [Fig. 7.13](#) while creating/modifying your study at <https://prolific.co>:

Note in the above that I have set the *participant*, *session* and *study_id* for our study using a URL query string. These values will be populated by Prolific when participants are sent to the study URL. Prolific will help you to format these correctly if you tick the *Include URL Parameters?* box which will bring up the following dialog. I’ve changed the default values that PsychoPy will use to store the variables (e.g. to be *participant* and *session* which are the default names for these in PsychoPy):

In each of the boxes in [Fig. 7.14](#), you can see the name that Prolific gives to this value (e.g. *PROLIFIC_PID*) and the name that we want PsychoPy to use to store it (e.g. *participant*).

STUDY LINK

How to record Prolific IDs

To link answers in your survey tool to participants in Prolific, you'll need to set up your survey tool to record our participants' unique Prolific IDs.

This enables you to match our participant [demographic data](#) with their answers. If you receive a poor quality submission, you can also [reject it in our platform](#).

What is the URL of your study?

[🔗](#) `http://run.pavlovia.org/myUserName/myStudyName/index.html?participant={{%PROLIFIC_PID%}}&study_id={{%STUDY_ID%}}&ses:`

How do you want to record Prolific IDs? (Select an option below for instructions)

I'll add a question in my study I'll use URL parameters I don't need to record these

To link answers in your survey tool to participants in Prolific, **you'll need to set up your survey tool** to record our participants' unique Prolific IDs.

Check out our [integration guide](#) instructions for the most commonly used survey tools.

Prolific ID Study ID Session ID [Configure parameters](#)

Fig. 7.13: Prolific settings for integration for PsychoPy

Setting the completion URL in PsychoPy

Do not show the completion code to your participants before they have completed your study. Displaying the completion code may result in data loss, since it encourages your participants to return to Prolific before they have completed your study. Instead copy the *Completion URL* from the main control panel above and paste that into the online tab for your PsychoPy *Experiment Settings* as in Fig. 7.15:

Integrating with Amazon's Mechanical Turk (MTurk)

Sorry, the documentation for this hasn't yet been written.

The features are in place to do this, and the general description of how to make it work are on the page [Recruiting participants and connecting with online services](#).

Warning: Using Mechanical truk (MTurk)

Note that Mechanical Turk was not designed with behavioural science in mind, but as a way for Amazon to test computing technologies in cases where a human was needed to push the buttons. They don't particularly care about the quality of this behavioral data as a result, nor about the ethics of the participants involved.

To get better quality data, and to run studies that your local ethical review board is less likely to be concerned about, then we would suggest you use a dedicated service like [Prolific Academic](#) instead.

URL Parameters ✕

Including URL parameters will allow you to automatically record the participant ID. The box below shows how they will be translated into your final study URL. [Read more about this here.](#)

```
http://run.pavlovia.org/myUserName/myStudyName/index.html?participant={{%PROLIFIC_PID%}}&study_id={{%STUDY_ID%}}&session={{%SESSION_ID%}}
```

PROLIFIC PID

STUDY ID

SESSION ID

Fig. 7.14: Prolific settings (inserting parameters dialog box)

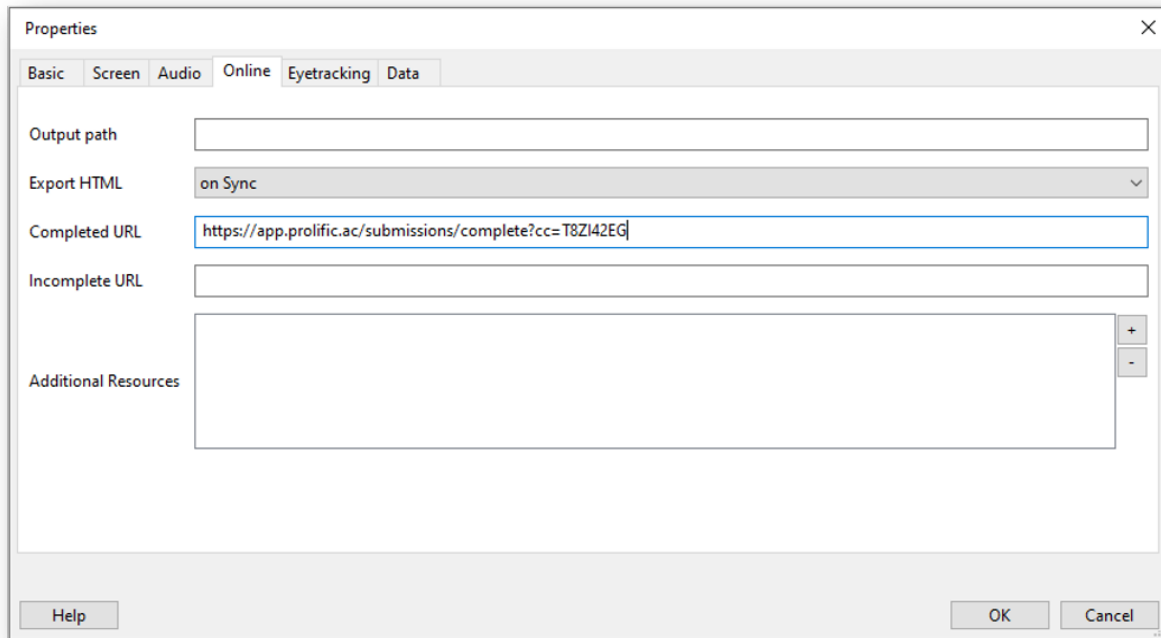


Fig. 7.15: The completion URL pasted into PsychoPy Experiment Settings

Daisy-chaining with Qualtrics

Sorry, the documentation for this hasn't yet been written.

The features are in place to do this, and the general description of how to make it work are on the page *Recruiting participants and connecting with online services*.

The general principle

All the systems below use the same general principle to connect the different services:

1. the recruiting system needs the URL of your study to send participants there. It probably needs to add the participant ID to that URL so that your study can store that information and (potentially) send it back to the recruiting system
2. at the end of the study the participant should be redirected back to the recruiting system so they can be credited with completing your task

Step 1. is obviously fairly easy if you know how to use the recruiting system. There will be a place somewhere in that system for you to enter the URL of the study being run. The key part is how to provide and store the participant/session ID, as described below.

Passing in a participant/session ID

PsychoPy experiments bring up a dialog box at the start of the study to collect information about a run, which usually includes information about the participant. You can adjust the fields of that box in the Experiment Settings dialog box



but usually there is a *participant* field (and we recommend you keep that!)

However, any variables can be passed to the experiment using the URL instead of the dialog box, and this is how you would typically pass the participant ID to your study. This is done by using “Query strings” which are a common part of online web addresses.

If your experiment has the address:

```
https://run.pavlovia.org/yourUsername/yourStudyName/index.html
```

then this URL will run the same study but with the *participant* variable set to be *10101010*:

```
https://run.pavlovia.org/yourUsername/yourStudyName/index.html?participant=10101010
```


and if you want two variables to be set then you can use *?* for the first and *&* for each subsequent. For instance this would set a *participant* as well as a *group* variable:

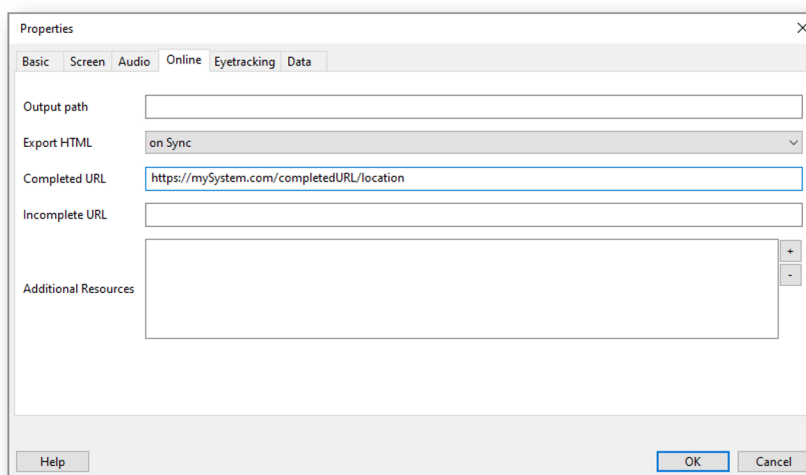
```
https://run.pavlovia.org/yourUsername/yourStudyName/index.html?participant=10101010&group=A
```

If you want to use that variable within your study you can do so using *expInfo*. For instance you could set a thank you message with this JavaScript code in your study:

```
msg = "Thanks, you're done. Your ID is " + expInfo['participant'];
```

Redirecting at the end of the study

This is really simple. In the Experiment Settings dialog again  you can select the *Online* tab and that has a setting to provide a link for completed and failed-to-complete participants:

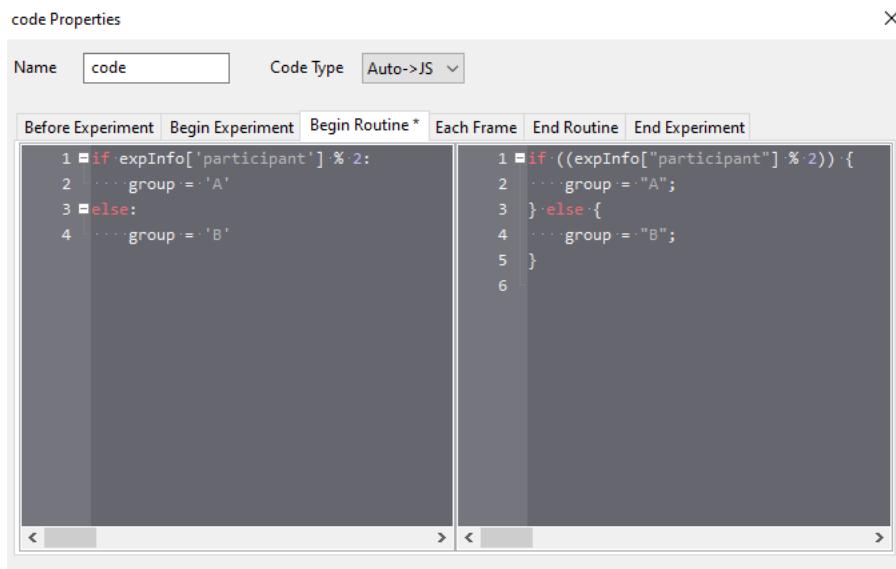


7.1.5 Counterbalancing online

If you are manually recruiting your participants (i.e. sending out your experiment URL to a unique population or group) the methods described in *Blocks of trials and counterbalancing* will also work online, and can be used in the same way. However, if you have your experiment URL advertised on a recruitment website, it could be that 10s or hundreds of participants click your link. Manually assigning participants and keeping track of participant groupings in these situations is going to be difficult. So, what do we do?

At the moment we don't have an internal method for keeping track of how many participants have already completed your task (and this is needed for counterbalancing). However, some core contributors have developed some excellent tools to help out, in particular this tool developed by [Wakefield Morys Carter](#) that generates sequential participant IDs for your task. Although not a counterbalance tool per se, we can use this to assign out participants to specific groups.

Add a code component to the beginning of your task that looks something like this:



The screenshot shows the 'code Properties' window in PsychoPy Builder. The 'Name' field is set to 'code' and the 'Code Type' is 'Auto->JS'. The code is organized into a timeline with tabs: 'Before Experiment', 'Begin Experiment', 'Begin Routine *', 'Each Frame', 'End Routine', and 'End Experiment'. Two code snippets are visible in the editor:

```

1 if expInfo['participant'] % 2:
2     ... group = 'A'
3 else:
4     ... group = 'B'

```

```

1 if ((expInfo["participant"] % 2)):{
2     ... group = "A";
3 } else {
4     ... group = "B";
5 }
6

```

Here we are checking if your participant ID is divisible by 2 (i.e. odd or even) and creating the variable 'group' using that. We would then use the same methods outlined previously in *Blocks of trials and counterbalancing* except this time we replace any instance of: `expInfo['group']` with `group`

So the conditions files used is selected based on the participant ID!

7.1.6 How does it work?

The first stage of this is that there is now a JavaScript library, [PsychoJS](#), that mirrors the PsychoPy Python library classes and functions.

PsychoPy Builder is effectively just writing a script for you based on the visual representation of your study so the new feature is for it simply to write a html/JavaScript/PsychoJS page instead.

Modern browsers are remarkably powerful. Most browsers released since 2011 have allowed HTML5 which supports more flexible rendering of web pages (images and text can be positioned precisely enough to run "proper" behavioural experiments). Since 2013 most have supported WebGL. That allows graphics to be rendered really quickly using "hardware acceleration" on your graphics card. The result is rich pages that can be updated very rapidly and can be forced to sync to screen refresh, which is critical for stimulus timing.

All this means we can do great things with online experiments that actually have good temporal precision!

The way it works is that you have a web page containing JavaScript (generated by PsychoPy Builder). You upload that to a web server. The participant of your study uses their web browser to visit the page you've created with a standard URL you send them.

Now, JavaScript executes on their computer (as opposed to scripts like PHP that operate on the server and aren't directly visible to the viewer/browser). In this case the PsychoJS script will present a dialog box at the start of the study to get the participant ID and any other basic information you need. While that dialog box is presented the script will be downloading all your stimuli and files to the local computer and storing them in memory. When all the necessary files are downloaded the participant can press "OK" and the experiment will start.

The experiment supports all the standard timing aspects of any PsychoPy Builder experiment; you can specify your stimuli in terms of time presented or number of screen refreshes etc (and the actual refresh rate of your participant's computer will be stored in your data file). When it's finished it saves the data into a comma-separated-value (CSV) file in the "data" folder on the web server. This looks very much like the standard CSV outputs of your same PsychoPy experiment run locally.

Not all components are currently supported. Keep an eye on the [onlineStatus](#) page to see what objects you can use already.

How does this compare with jsPsych?

In jsPsych you use one of the pre-programmed "types" of trial (like single stimulus or 2-alternative-forced-choice) and you have rather little flexibility over how that gets conducted. If you wanted to alter the positioning of the stimuli, for instance, in a 2-alternative-force-choice task or you wanted a stimulus to change in time (appear gradually or move location) then you would need to write a new trial "type" using raw javascript.

PsychoPy, by comparison, is designed to give you total flexibility. You decide what constitutes a "trial" and how things should operate in time. We think that control is very important to creating a wide range of studies.

7.1.7 Manual coding of PsychoJS studies

Note that PsychoJS is very much under development and all parts of the API are subject to change

Some people may want to write a JS script from scratch or convert their PsychoPy Python script into '**PsychoJS**'_. However, supporting this approach is beyond the scope of our documentation and our [forum](#).

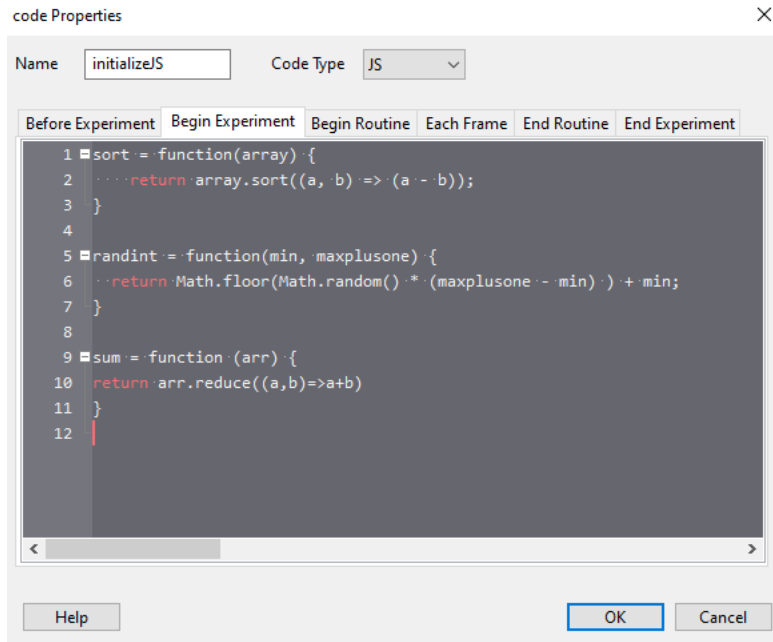
Working with JS Code Components

Code components can automatically convert Python to JavaScript. However, this doesn't always work. Below are some pointers to help you out:

- For common JS functions, see the [PsychoPy to JS crib sheet](#) by [Wakefield Morys-Carter](#)
- For finding out how to manipulate PsychoJS components via code, see the [PsychoJS API](#). The [tutorial_js_expose_psychajs](#) experiment shows how to expose PsychoJS objects to the web browser, so that you can access them via the browser console, and try things out in order to see what works (or not).
- If you're looking for a JS equivalent of a Python function, try searching 'JS equivalent/version of function X' on [stack overflow](#) or [Google](#)
- Still stuck? Try asking for help on the [forum](#). For giving researchers access to the repository of your experiment, see [Contributing an experiment to Pavlovia](#)

Adding JS functions

If you have a function you want to use, and you find the equivalent on the crib sheet or stack overflow, add an ‘initialization’ code component to the start of your experiment. Set code type to be ‘JS’ and copy and paste the function(s) you want there in the ‘Begin experiment’ tab. These functions will then be available to be called throughout the rest of the task.



Don't change the generated JS file

When you export an experiment to HTML from the PsychoPy builder, it generates a JS file. We recommend *not* to edit this JS file, for the reasons below:

- Changes you make in your .js file will not be reflected back in your builder file; it is a one way street.
- It becomes more difficult to sync your experiment with from the builder
- Researchers that would like to replicate your experiment but aren't very JavaScript-savvy might be better off using the PsychoPy Builder

The first generation of PsychoJS was realized by a [Wellcome Trust](#) grant, awarded in January 2018. to make online studies possible from . This is what we call PsychoPy3 - the 3rd major phase of PsychoPy's development.

COMMUNICATING WITH EXTERNAL HARDWARE USING PSYCHOPY

PsychoPy is able to communicate with a range of external hardware, like EEG recording devices and eye trackers.

This page provides step-by-step instructions on how to communicate with some of the more commonly used hardware. The page is being updated regularly so if you don't see your device listed here please do post in the forum as we keep an eye on commonly-faced issues (and solutions!) there.

8.1 Communicating with EEG

Before getting started with an EEG study in PsychoPy, we **highly** recommend reading relevant information on how to measure and understand *Timing Issues and synchronisation*. Although these guides will talk you through how to communicate with EEG hardware, they can really be used to communicate with any device that is connected via the same method:

- parallel
- serial
- egi
- [Communicating with Emotiv](#) please also see [this video tutorial](#).

Note: If you'd like to use a *Parallel Port* to **record** responses (for example from a button box) please read [this excellent thread](#) from our Discourse Forum user [jtseng](#).

8.2 Communicating with an eye-tracker

- eyetracking

8.3 Communicating with other devices

- fmri
- arduino
- To communicate with fNIRS, please watch [this super-clear video tutorial](#) from NIRx.

REFERENCE MANUAL (API)

Contents:

9.1 psychopy.core - basic functions (clocks etc.)

Basic functions, including timing, rush (imported), quit

class psychopy.core.Clock

A convenient class to keep track of time in your experiments. You can have as many independent clocks as you like (e.g. one to time responses, one to keep track of stimuli ...)

This clock is identical to the *MonotonicClock* except that it can also be reset to 0 or another value at any point.

add(*t*)

DEPRECATED: use `.addTime()` instead

This function adds time TO THE BASE (*t*0) which, counterintuitively, reduces the apparent time on the clock

addTime(*t*)

Add more time to the Clock/Timer

e.g.:

```
timer = core.Clock()
timer.add(5)
while timer.getTime() < 0:
    # do something
```

reset(*newT=0.0*)

Reset the time on the clock. With no args time will be set to zero. If a float is received this will be the new time on the clock

class psychopy.core.CountdownTimer(*start=0*)

Similar to a *Clock* except that time counts down from the time of last reset.

Parameters **start** (*float* or *int*) – Starting time in seconds to countdown on.

Examples

Create a countdown clock with a 5 second duration:

```
timer = core.CountdownTimer(5)
while timer.getTime() > 0: # after 5s will become negative
    # do stuff
```

getTime()

Returns the current time left on this timer in seconds with sub-ms precision (*float*).

reset (*t=None*)

Reset the time on the clock.

Parameters *t* (*float, int or None*) – With no args (*None*), time will be set to the time used for last reset (or start time if no previous resets). If a number is received, this will be the new time on the clock.

class psychopy.core.**MonotonicClock** (*start_time=None*)

A convenient class to keep track of time in your experiments using a sub-millisecond timer.

Unlike the *Clock* this cannot be reset to arbitrary times. For this clock *t=0* always represents the time that the clock was created.

Don't confuse this *class* with *core.monotonicClock* which is an *instance* of it that got created when PsychoPy.core was imported. That clock instance is deliberately designed always to return the time since the start of the study.

Version Notes: This class was added in PsychoPy 1.77.00

getLastResetTime ()

Returns the current offset being applied to the high resolution timebase used by Clock.

getTime (*applyZero=True*)

Returns the current time on this clock in secs (sub-ms precision).

If applying zero then this will be the time since the clock was created (typically the beginning of the script).

If not applying zero then it is whatever the underlying clock uses as its base time but that is system dependent. e.g. can be time since reboot, time since Unix Epoch etc

class psychopy.core.**StaticPeriod** (*screenHz=None, win=None, name='StaticPeriod'*)

A class to help insert a timing period that includes code to be run.

Parameters

- **screenHz** (*int or None*) –
- **frame rate of the monitor (leave as None if you (the) – don't want this accounted for)**
- **win** (*Window*) – If a *Window* is given then *StaticPeriod* will also pause/restart frame interval recording.
- **name** (*str*) – Give this *StaticPeriod* a name for more informative logging messages.

Examples

Typical usage for the static period:

```
fixation.draw()
win.flip()
ISI = StaticPeriod(screenHz=60)
ISI.start(0.5) # start a period of 0.5s
stim.image = 'largeFile.bmp' # could take some time
ISI.complete() # finish the 0.5s, taking into account one 60Hz frame

stim.draw()
win.flip() # the period takes into account the next frame flip
# time should now be at exactly 0.5s later than when ISI.start()
# was called
```

complete()

Completes the period, using up whatever time is remaining with a call to *wait()*.

Returns 1 for success, 0 for fail (the period overran).

Return type float

start (*duration*)

Start the period. If this is called a second time, the timer will be reset and starts again

Parameters *duration* (*float* or *int*) – The duration of the period, in seconds.

`psychopy.core.getAbsTime()`

Get the absolute time.

This uses the same clock-base as the other timing features, like *getTime()*. The time (in seconds) ignores the time-zone (like *time.time()* on linux). To take the timezone into account, use *int(time.mktime(time.gmtime()))*.

Absolute times in seconds are especially useful to add to generated file names for being unique, informative (= a meaningful time stamp), and because the resulting files will always sort as expected when sorted in chronological, alphabetical, or numerical order, regardless of locale and so on.

Version Notes: This method was added in PsychoPy 1.77.00

Returns Absolute Unix time (i.e., whole seconds elapsed since Jan 1, 1970).

Return type float

`psychopy.core.getTime` (*applyZero=True*)

Get the current time since `psychopy.core` was loaded.

Version Notes: Note that prior to PsychoPy 1.77.00 the behaviour of `getTime()` was platform dependent (on OSX and linux it was equivalent to `psychopy.core.getAbsTime()` whereas on windows it returned time since loading of the module, as now)

`psychopy.core.wait` (*secs*, *hogCPUperiod=0.2*)

Wait for a given time period.

If *secs=10* and *hogCPU=0.2* then for 9.8s Python's *time.sleep* function will be used, which is not especially precise, but allows the cpu to perform housekeeping. In the final *hogCPUperiod* the more precise method of constantly polling the clock is used for greater precision.

If you want to obtain key-presses during the wait, be sure to use `pyglet` and to `hogCPU` for the entire time, and then call `psychopy.event.getKeys()` after calling `wait()`

If you want to suppress checking for `pyglet` events during the wait, do this once:

```
core.checkPygletDuringWait = False
```

and from then on you can do:

```
core.wait(sec)
```

This will preserve terminal-window focus during command line usage.

Parameters

- **secs** (*float or int*) –
- **hogCPUperiod** (*float or int*) –

9.2 psychopy.clock - Clocks and timers

Created on Tue Apr 23 11:28:32 2013

Provides the high resolution timebase used by psychopy, and defines some time related utility Classes.

Moved functionality from core.py so a common code base could be used in core.py and logging.py; vs. duplicating the getTime and Clock logic.

@author: Sol @author: Jon

class psychopy.clock.Clock

A convenient class to keep track of time in your experiments. You can have as many independent clocks as you like (e.g. one to time responses, one to keep track of stimuli ...)

This clock is identical to the *MonotonicClock* except that it can also be reset to 0 or another value at any point.

add (*t*)

DEPRECATED: use .addTime() instead

This function adds time TO THE BASE (*t0*) which, counterintuitively, reduces the apparent time on the clock

addTime (*t*)

Add more time to the Clock/Timer

e.g.:

```
timer = core.Clock()
timer.add(5)
while timer.getTime() < 0:
    # do something
```

reset (*newT=0.0*)

Reset the time on the clock. With no args time will be set to zero. If a float is received this will be the new time on the clock

class psychopy.clock.CountdownTimer (*start=0*)

Similar to a *Clock* except that time counts down from the time of last reset.

Parameters **start** (*float or int*) – Starting time in seconds to countdown on.

Examples

Create a countdown clock with a 5 second duration:

```
timer = core.CountdownTimer(5)
while timer.getTime() > 0: # after 5s will become negative
    # do stuff
```

getTime()

Returns the current time left on this timer in seconds with sub-ms precision (*float*).

reset (*t=None*)

Reset the time on the clock.

Parameters *t* (*float, int or None*) – With no args (*None*), time will be set to the time used for last reset (or start time if no previous resets). If a number is received, this will be the new time on the clock.

class psychopy.clock.**MonotonicClock** (*start_time=None*)

A convenient class to keep track of time in your experiments using a sub-millisecond timer.

Unlike the *Clock* this cannot be reset to arbitrary times. For this clock *t=0* always represents the time that the clock was created.

Don't confuse this *class* with *core.monotonicClock* which is an *instance* of it that got created when PsychoPy.core was imported. That clock instance is deliberately designed always to return the time since the start of the study.

Version Notes: This class was added in PsychoPy 1.77.00

getLastResetTime ()

Returns the current offset being applied to the high resolution timebase used by Clock.

getTime (*applyZero=True*)

Returns the current time on this clock in secs (sub-ms precision).

If applying zero then this will be the time since the clock was created (typically the beginning of the script).

If not applying zero then it is whatever the underlying clock uses as its base time but that is system dependent. e.g. can be time since reboot, time since Unix Epoch etc

class psychopy.clock.**StaticPeriod** (*screenHz=None, win=None, name='StaticPeriod'*)

A class to help insert a timing period that includes code to be run.

Parameters

- **screenHz** (*int or None*) –
- **frame rate of the monitor (leave as None if you (the) –** don't want this accounted for)
- **win** (*Window*) – If a *Window* is given then *StaticPeriod* will also pause/restart frame interval recording.
- **name** (*str*) – Give this *StaticPeriod* a name for more informative logging messages.

Examples

Typical usage for the static period:

```
fixation.draw()
win.flip()
ISI = StaticPeriod(screenHz=60)
ISI.start(0.5) # start a period of 0.5s
stim.image = 'largeFile.bmp' # could take some time
ISI.complete() # finish the 0.5s, taking into account one 60Hz frame

stim.draw()
win.flip() # the period takes into account the next frame flip
# time should now be at exactly 0.5s later than when ISI.start()
# was called
```

complete()

Completes the period, using up whatever time is remaining with a call to *wait()*.

Returns 1 for success, 0 for fail (the period overran).

Return type float

start (*duration*)

Start the period. If this is called a second time, the timer will be reset and starts again

Parameters *duration* (*float* or *int*) – The duration of the period, in seconds.

`psychopy.clock.getAbsTime()`

Get the absolute time.

This uses the same clock-base as the other timing features, like *getTime()*. The time (in seconds) ignores the time-zone (like *time.time()* on linux). To take the timezone into account, use *int(time.mktime(time.gmtime()))*.

Absolute times in seconds are especially useful to add to generated file names for being unique, informative (= a meaningful time stamp), and because the resulting files will always sort as expected when sorted in chronological, alphabetical, or numerical order, regardless of locale and so on.

Version Notes: This method was added in PsychoPy 1.77.00

Returns Absolute Unix time (i.e., whole seconds elapsed since Jan 1, 1970).

Return type float

`psychopy.clock.getTime()`

Copyright (c) 2018 Mario Kleiner. Licensed under MIT license.

For detailed help on a subfunction SUBFUNCTIONNAME, type `GetSecs('SUBFUNCTIONNAME?')` i.e. the name with a question mark appended. E.g., for detailed help on the subfunction called `Version`, type this: `GetSecs('Version?')`

`[GetSecsTime, WallTime, syncErrorSecs, MonotonicTime] = GetSecs('AllClocks' [, maxError=0.000020]);`

`psychopy.clock.wait` (*secs*, *hogCPUperiod=0.2*)

Wait for a given time period.

If *secs=10* and *hogCPU=0.2* then for 9.8s Python's *time.sleep* function will be used, which is not especially precise, but allows the cpu to perform housekeeping. In the final *hogCPUperiod* the more precise method of constantly polling the clock is used for greater precision.

If you want to obtain key-presses during the wait, be sure to use `pyglet` and to `hogCPU` for the entire time, and then call `psychopy.event.getKeys()` after calling `wait()`

If you want to suppress checking for `pyglet` events during the wait, do this once:

```
core.checkPygletDuringWait = False
```

and from then on you can do:

```
core.wait(sec)
```

This will preserve terminal-window focus during command line usage.

Parameters

- **secs** (*float or int*) –
- **hogCPUperiod** (*float or int*) –

9.3 psychopy.visual - many visual stimuli

9.3.1 Aperture

Attributes

<i>Aperture</i> (win[, size, pos, anchor, ori, ...])	Restrict a stimulus visibility area to a basic shape or list of vertices.
<i>Aperture.size</i>	Set the size (diameter) of the Aperture.
<i>Aperture.pos</i>	Set the pos (centre) of the Aperture.
<i>Aperture.ori</i>	Set the orientation of the Aperture.
<i>Aperture.inverted</i>	True / False.
<i>Aperture.name</i>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<i>Aperture.autoLog</i>	Whether every change in this stimulus should be auto logged.

```
class psychopy.visual.Aperture (win, size=1, pos=0, 0, anchor=None, ori=0, nVert=120,
                                shape='circle', inverted=False, units=None, name=None, au-
                                toLog=None)
```

Restrict a stimulus visibility area to a basic shape or list of vertices.

When enabled, any drawing commands will only operate on pixels within the Aperture. Once disabled, subsequent draw operations affect the whole screen as usual.

Supported shapes:

- ‘square’, ‘triangle’, ‘circle’ or *None*: a polygon with appropriate nVerts will be used (120 for ‘circle’)
- integer: a polygon with that many vertices will be used
- list or numpy array (Nx2): it will be used directly as the vertices to a *ShapeStim*
- a filename then it will be used to load and image as a *ImageStim*. Note that transparent parts in the image (e.g. in a PNG file) will not be included in the mask shape. The color of the image will be ignored.

See demos/stimuli/aperture.py for example usage

Author 2011, Yuri Spitsyn 2011, Jon Peirce added units options, Jeremy Gray added shape & orientation 2014, Jeremy Gray added .contains() option 2015, Thomas Emmerling added ImageStim option

`_reset ()`

Internal method to rebuild the shape - shouldn't be called by the user. You have to explicitly turn resetting off by setting `self._needReset = False`

`_updateVertices ()`

Sets `Stim.verticesPix` and `._borderPix` from `pos`, `size`, `ori`, `flipVert`, `flipHoriz`

property `anchor`

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`contains (x, y=None, units=None)`

Returns True if a point `x,y` is inside the stimulus' border.

Can accept variety of input options:

- two separate args, `x` and `y`
- one arg (list, tuple or array) containing two vals (`x,y`)
- **an object with a `getPos()` method that returns `x,y`, such** as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the *size*, *position (pos)*, and *orientation (ori)* settings (and by the *vertices* for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

`disable ()`

Use `Aperture.enabled = False` instead.

`enable ()`

Use `Aperture.enabled = True` instead.

`enabled`

True / False. Enable or disable the aperture. Determines whether it is used in future drawing operations.

NB. The Aperture is enabled by default, when created.

property `flip`

`invert ()`

Use `Aperture.inverted = True` instead.

`inverted`

True / False. Set to true to invert the aperture. A non-inverted aperture masks everything BUT the selected shape. An inverted aperture masks the selected shape.

NB. The Aperture is not inverted by default, when created.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

ori

Set the orientation of the Aperture.

This essentially controls a *ShapeStim* so see documentation for ShapeStim.ori.

Operations supported here as well as ShapeStim.

Use setOri() if you want to control logging and resetting.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

Set the pos (centre) of the Aperture. *Operations* supported.

This essentially controls a *ShapeStim* so see documentation for ShapeStim.pos.

Operations supported here as well as ShapeStim.

Use setPos() if you want to control logging and resetting.

property posPix

The position of the aperture in pixels

setAnchor (*value, log=None*)

setAutoDraw (*value, log=None*)

Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True, log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setOri (*ori, needReset=True, log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setPos (*pos, needReset=True, log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setSize (*size, needReset=True, log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

property size

Set the size (diameter) of the Aperture.

This essentially controls a *ShapeStim* so see documentation for ShapeStim.size.

Operations supported here as well as ShapeStim.

Use setSize() if you want to control logging and resetting.

property sizePix

The size of the aperture in pixels

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

9.3.2 BoundingBox

Attributes

<i>BoundingBox</i> ([extents])	Class for representing object bounding boxes.
--------------------------------	---

Details

class psychopy.visual.**BoundingBox** (*extents=None*)

Class for representing object bounding boxes.

A bounding box is a construct which represents a 3D rectangular volume about some pose, defined by its minimum and maximum extents in the reference frame of the pose. The axes of the bounding box are aligned to the axes of the world or the associated pose.

Bounding boxes are primarily used for visibility testing; to determine if the extents of an object associated with a pose (eg. the vertices of a model) falls completely outside of the viewing frustum. If so, the model can be culled during rendering to avoid wasting CPU/GPU resources on objects not visible to the viewer.

_computeCorners ()

Compute the corners of the bounding box.

These values are cached to speed up computations if extents hasn't been updated.

clear ()

Clear a bounding box, invalidating it.

property extents

fit (*verts*)

Fit the bounding box to vertices.

property isValid

True if the bounding box is valid.

9.3.3 BoxStim

Attributes

<i>BoxStim</i> (win[, size, flipFaces, pos, ori, ...])	Class for drawing 3D boxes.
--	-----------------------------

Details

```
class psychopy.visual.BoxStim(win, size=0.5, 0.5, 0.5, flipFaces=False, pos=0.0, 0.0, 0.0,
                               ori=0.0, 0.0, 0.0, 1.0, color=0.0, 0.0, 0.0, colorSpace='rgb', contrast=1.0, opacity=1.0, useMaterial=None, textureScale=None,
                               name="", autoLog=True)
```

Class for drawing 3D boxes.

Draws a rectangular box with dimensions specified by *size* (length, width, height) in scene units.

Calling the *draw* method will render the box to the current buffer. The render target (FBO or back buffer) must have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting *useLights=True* before drawing the stimulus).

Warning: This class is experimental and may result in undefined behavior.

Parameters

- **win** (*~psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- **size** (*tuple or float*) – Dimensions of the mesh. If a single value is specified, the box will be a cube. Provide a tuple of floats to specify the width, length, and height of the box (eg. *size=(0.2, 1.3, 2.1)*) in scene units.
- **flipFaces** (*bool, optional*) – If *True*, normals and face windings will be set to point inward towards the center of the box. Texture coordinates will remain the same. Default is *False*.
- **pos** (*array_like*) – Position vector $[x, y, z]$ for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion $[x, y, z, w]$ where x, y, z are imaginary and w is real. If you prefer specifying rotations in axis-angle format, call *setOriAxisAngle* after initialization.
- **useMaterial** (*PhongMaterial, optional*) – Material to use. The material can be configured by accessing the *material* attribute after initialization. If not material is specified, the diffuse and ambient color of the shape will track the current color specified by *glColor*.
color : *array_like* Diffuse and ambient color of the stimulus if *useMaterial* is not specified. Values are with respect to *colorSpace*.
- **colorSpace** (*str*) – Colorspace of *color* to use.
- **contrast** (*float*) – Contrast of the stimulus, value modulates the *color*.
- **opacity** (*float*) – Opacity of the stimulus ranging from 0.0 to 1.0. Note that transparent objects look best when rendered from farthest to nearest.
- **textureScale** (*array_like or float, optional*) – Scaling factors for texture coordinates (sx, sy). By default, a factor of 1 will have the entire texture cover the surface of the mesh. If a single number is provided, the texture will be scaled uniformly.

- **name** (*str*) – Name of this object for logging purposes.
- **autoLog** (*bool*) – Enable automatic logging on attribute changes.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

_createVAO (*vertices, textureCoords, normals, faces*)

Create a vertex array object for handling vertex attribute data.

_getDesiredRGB (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires *self.rgb*, *self.colorSpace* and *self.contrast*

_selectWindow (*win*)

Switch drawing to the specified window. Calls the window's *_setCurrent()* method which handles the switch.

_updateList ()

The user shouldn't need this method since it gets called after every call to *.set()* Chooses between using and not using shaders each call.

property anchor

property backColor

Alternative way of setting *fillColor*

property backColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property borderColor

property borderColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default *colorSpace* for the stimulus is used (defined during initialisation).

Please note that changing *colorSpace* does not change stimulus parameters. Thus you usually want to specify *colorSpace* before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')
```

(continues on next page)

(continued from previous page)

```
# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

draw (*win=None*)

Draw the stimulus.

This should work for stimuli using a single VAO and material. More complex stimuli with multiple materials should override this method to correctly handle that case.

Parameters *win* (~*psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

getOri ()

getOriAxisAngle (degrees=True)

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the *ori* quaternion to and axis-angle representation.

Parameters **degrees** (*bool, optional*) – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

Returns Axis [*rx, ry, rz*] and angle.

Return type `tuple`

getPos ()

getRayIntersectBounds (*rayOrig*, *rayDir*)

Get the point which a ray intersects the bounding box of this mesh.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.

Returns Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type `tuple`

property height

isVisible ()

Check if the object is visible to the observer.

Test if a pose's bounding box or position falls outside of an eye's view frustum.

Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e. triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

Returns *True* if the object's bounding box is visible.

Return type `bool`

Examples

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```
if myStim.isVisible():
    myStim.draw()
```

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

property ori

Orientation quaternion (X, Y, Z, W).

property pos

Position vector (X, Y, Z).

setAnchor (*value*, *log=None*)

setBackColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setBackRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setBorderRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor (*color, colorSpace=None, operation="", log=None*)

setContrast (*newContrast, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setFillColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setFillRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *foreColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setForeRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor (*color, colorSpace=None, operation="", log=None*)

setLineRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOri (*ori*)

setOriAxisAngle (*axis, angle, degrees=True*)

Set the orientation of the 3D stimulus using an *axis* and *angle*. This sets the quaternion at *ori*.

Parameters

- **axis** (*array_like*) – Axis of rotation [rx, ry, rz].
- **angle** (*float*) – Angle of rotation.
- **degrees** (*bool, optional*) – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

setPos (*pos*)

setRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

property size

property thePose

The pose of the rigid body. This is a class which has *pos* and *ori* attributes.

units

None, 'norm', 'cm', 'deg', 'degFlat', 'degFlatPos', or 'pix'

If None then the current units of the *Window* will be used. See *Units for the window and stimuli* for explanation of other options.

Note that when you change units, you don't change the stimulus parameters and it is likely to change appearance. Example:

```
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5)

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

property vertices

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.4 BufferImageStim

Attributes

<i>BufferImageStim</i> (win[, buffer, rect, ...])	Take a “screen-shot”, save as an ImageStim (RBGA object).
<i>BufferImageStim.win</i>	The <i>Window</i> object in which the stimulus will be rendered by default.
<i>BufferImageStim.mask</i>	The alpha mask that can be used to control the outer shape of the stimulus
<i>BufferImageStim.units</i>	
<i>BufferImageStim.pos</i>	The position of the center of the stimulus in the stimulus <i>units</i>
<i>BufferImageStim.ori</i>	The orientation of the stimulus (in degrees).

continues on next page

Table 9.4 – continued from previous page

<code>BufferImageStim.size</code>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<code>BufferImageStim.contrast</code>	A value that is simply multiplied by the color.
<code>BufferImageStim.color</code>	Alternative way of setting <i>foreColor</i> .
<code>BufferImageStim.colorSpace</code>	The name of the color space currently being used
<code>BufferImageStim.opacity</code>	Determines how visible the stimulus is relative to background.
<code>BufferImageStim.interpolate</code>	Whether to interpolate (linearly) the texture in the stimulus.
<code>BufferImageStim.name</code>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<code>BufferImageStim.autoLog</code>	Whether every change in this stimulus should be auto logged.
<code>BufferImageStim.draw([win])</code>	Draws the <code>BufferImage</code> on the screen, similar to <code>ImageStim.draw()</code> .
<code>BufferImageStim.autoDraw</code>	Determines whether the stimulus should be automatically drawn on every frame flip.

Details

```
class psychopy.visual.BufferImageStim(win, buffer='back', rect=- 1, 1, 1, - 1,
                                       sqPower2=False, stim=(), interpolate=True,
                                       flipHoriz=False, flipVert=False, mask='None',
                                       pos=0, 0, name=None, autoLog=None)
```

Take a “screen-shot”, save as an `ImageStim` (RGBA object).

The screen-shot is a single collage image composed of static elements that you can treat as being a single stimulus. The screen-shot can be of the visible screen (front buffer) or hidden (back buffer).

`BufferImageStim` aims to provide fast rendering, while still allowing dynamic orientation, position, and opacity. It’s fast to draw but slower to init (same as an `ImageStim`).

You specify the part of the screen to capture (in norm units), and optionally the stimuli themselves (as a list of items to be drawn). You get a screenshot of those pixels. If your OpenGL does not support arbitrary sizes, the image will be larger, using square powers of two if needed, with the excess image being invisible (using alpha). The aim is to preserve the buffer contents as rendered.

Checks for OpenGL 2.1+, or uses square-power-of-2 images.

Example:

```
# define lots of stimuli, make a list:
mySimpleImageStim = ...
myTextStim = ...
stimList = [mySimpleImageStim, myTextStim]

# draw stim list items & capture (slow; see EXP log for times):
screenshot = visual.BufferImageStim(myWin, stim=stimList)

# render to screen (very fast, except for the first draw):
while <conditions>:
    screenshot.draw() # fast; can vary .ori, .pos, .opacity
    other_stuff.draw() # dynamic
    myWin.flip()
```

See coder Demos > stimuli > `bufferImageStim.py` for a demo, with timing stats.

Author

- 2010 Jeremy Gray, with on-going fixes

Parameters

buffer : the screen buffer to capture from, default is 'back' (hidden). 'front' is the buffer in view after `win.flip()`

rect : a list of edges [left, top, right, bottom] defining a screen rectangle which is the area to capture from the screen, given in norm units. default is fullscreen: [-1, 1, 1, -1]

stim : a list of item(s) to be drawn to the back buffer (in order). The back buffer is first cleared (without the win being flip(ed)), then stim items are drawn, and finally the buffer (or part of it) is captured. Each item needs to have its own `.draw()` method, and have the same window as win.

interpolate : whether to use interpolation (default = True, generally good, especially if you change the orientation)

sqPower2 :

- False (default) = use rect for size if OpenGL = 2.1+
- True = use square, power-of-two image sizes

flipHoriz : horizontally flip (mirror) the captured image, default = False

flipVert : vertically flip (mirror) the captured image; default = False

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use `obj._foreColor.rgb`

Type DEPRECATED

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`

`_createTexture(tex, id, pixFormat, stim, res=128, maskParams=None, forcePOW2=True, dataType=None, wrapping=True)`

Create a new OpenGL 2D image texture.

Parameters

- **tex** (*Any*) – Texture data. Value can be anything that resembles image data.
- **id** (*int* or *GLint*) – Texture ID.
- **pixFormat** (*GLenum* or *int*) – Pixel format to use, values can be *GL_ALPHA* or *GL_RGB*.
- **stim** (*Any*) – Stimulus object using the texture.
- **res** (*int*) – The resolution of the texture (unless a bitmap image is used).
- **maskParams** (*dict* or *None*) – Additional parameters to configure the mask used with this texture.
- **forcePOW2** (*bool*) – Force the texture to be stored in a square memory area. For grating stimuli (anything that needs multiple cycles) *forcePOW2* should be set to be *True*. Otherwise the wrapping of the texture will not work.
- **dataType** (*class:~pyglet.gl.GGLenum*, *int* or *None*) – *None*, *GL_UNSIGNED_BYTE*, *GL_FLOAT*. Only affects image files (numpy arrays will be float).

- **wrapping** (*bool*) – Enable wrapping of the texture. A texture will be set to repeat (or tile).

__getDesiredRGB (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

__getPolyAsRendered ()

DEPRECATED. Return a list of vertices as rendered.

__movieFrameToTexture (*movieSrc*)

Convert a movie frame to a texture and use it.

This method is used internally to copy pixel data from a camera object into a texture. This enables the *ImageStim* to be used as a ‘viewfinder’ of sorts for the camera to view a live video stream on a window.

Parameters movieSrc (~*psychopy.hardware.camera.Camera*) – Movie source object.

__selectWindow (*win*)

Switch drawing to the specified window. Calls the window’s `__setCurrent()` method which handles the switch.

__set (*attrib, val, op="", log=None*)

DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

__updateList ()

The user shouldn’t need this method since it gets called after every call to `.set()` Chooses between using and not using shaders each call.

__updateListShaders ()

The user shouldn’t need this method since it gets called after every call to `.set()` Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the `.set()` command

__updateVertices ()

Sets `Stim.verticesPix` and `._borderPix` from `pos`, `size`, `ori`, `flipVert`, `flipHoriz`

property anchor

autoDraw

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

autoLog

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren’t likely to be useful.

property backColor

Alternative way of setting `fillColor`

property backColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property borderColor

property borderColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

clearTextures ()

Clear all textures associated with the stimulus.

As of v1.61.00 this is called automatically during garbage collection of your stimulus, so doesn't need calling explicitly by the user.

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains (x, y=None, units=None)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such** as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw (*win=None*)

Draws the `BufferImage` on the screen, similar to `ImageStim.draw()`. Allows dynamic position, size, rotation, mirroring, and opacity. Limitations / bugs: not sure what happens with shaders and `self._updateList()`

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as `True` to flip or `False` to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

flipHoriz

If set to `True` then the image will be flipped horizontally (left-to-right). Note that this is relative to the original image, not relative to the current state.

flipVert

If set to `True` then the image will be flipped vertically (left-to-right). Note that this is relative to the original image, not relative to the current state.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or** other *Color spaces*. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

property height

image

The image file to be presented (most formats supported).

This can be a path-like object to an image file, or a numpy array of shape [H, W, C] where C are channels. The third dim will usually have length 1 (defining an intensity-only image), 3 (defining an RGB image) or 4 (defining an RGBA image).

If passing a numpy array to the image attribute, the size attribute of ImageStim must be set explicitly.

interpolate

Whether to interpolate (linearly) the texture in the stimulus.

If set to False then nearest neighbour will be used when needed, otherwise some form of interpolation will be used.

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

mask

The alpha mask that can be used to control the outer shape of the stimulus

- **None**, 'circle', 'gauss', 'raisedCos'
- or the name of an image file (most formats supported)
- or a numpy array (1xN or NxN) ranging -1:1

maskParams

Various types of input. Default to *None*.

This is used to pass additional parameters to the mask if those are needed.

- **For 'gauss' mask, pass dict {'sd': 5} to control** standard deviation.
- **For the 'raisedCos' mask, pass a dict: {'fringeWidth':0.2}**, where 'fringeWidth' is a parameter (float, 0-1), determining the proportion of the patch that will be blurred by the raised cosine edge.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called "unnamed <type>", e.g. `visual.TextStim(win)` will be called "unnamed TextStim" in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, `shapeContains.py`

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAnchor (*value*, *log=None*)

setAutoDraw (*value*, *log=None*)

Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setBackColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setBackRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setBorderRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setContrast (*newContrast*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setDKL (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setFillColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setFillRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setFlipHoriz (*newVal=True, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setFlipVert (*newVal=True, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setForeColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setImage (*value, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setLMS (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor (*color, colorSpace=None, operation="", log=None*)

setLineRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setMask (*value, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setOpacity (*newOpacity, operation="", log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos (*newPos, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize (*newSize, operation="", units=None, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at `stim._sizeRendered`

texRes

Power-of-two int. Sets the resolution of the mask and texture. `texRes` is overridden if an array or image is provided as mask.

Operations supported.

property units

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (`win1`, `win2` and `stim`):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.5 psychopy.visual.Circle

Stimulus class for drawing circles.

Overview

<code>Circle(win[, radius, edges, units, ...])</code>	Creates a Circle with a given radius as a special case of a <code>ShapeStim</code>
<code>Circle.radius</code>	float, int, tuple, list or 2x1 array Radius of the Polygon (distance from the center to the corners).
<code>Circle.edges</code>	Number of edges of the polygon.
<code>Circle.units</code>	
<code>Circle.lineWidth</code>	Width of the line in pixels .
<code>Circle.lineColor</code>	Alternative way of setting <code>borderColor</code> .
<code>Circle.lineColorSpace</code>	Deprecated, please use <code>colorSpace</code> to set color space for the entire object
<code>Circle.fillColor</code>	Set the fill color for the shape.
<code>Circle.fillColorSpace</code>	Deprecated, please use <code>colorSpace</code> to set color space for the entire object.
<code>Circle.pos</code>	The position of the center of the stimulus in the stimulus <i>units</i>
<code>Circle.size</code>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<code>Circle.ori</code>	The orientation of the stimulus (in degrees).
<code>Circle.opacity</code>	Determines how visible the stimulus is relative to background.
<code>Circle.contrast</code>	A value that is simply multiplied by the color.
<code>Circle.depth</code>	DEPRECATED, depth is now controlled simply by drawing order.
<code>Circle.interpolate</code>	If <i>True</i> the edge of the line will be anti-aliased.
<code>Circle.lineRGB</code>	Legacy property for setting the border color of a stimulus in RGB, instead use <code>obj._borderColor.rgb</code>
<code>Circle.fillRGB</code>	Legacy property for setting the fill color of a stimulus in RGB, instead use <code>obj._fillColor.rgb</code>
<code>Circle.name</code>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<code>Circle.autoLog</code>	Whether every change in this stimulus should be auto logged.
<code>Circle.autoDraw</code>	Determines whether the stimulus should be automatically drawn on every frame flip.
<code>Circle.color</code>	Set the color of the shape.
<code>Circle.colorSpace</code>	The name of the color space currently being used

Details

```
class psychopy.visual.circle.Circle (win, radius=0.5, edges=32, units="", lineWidth=1.5, lineColor=None, lineColorSpace=None, fillColor='white', fillColorSpace=None, pos=0, 0, size=1.0, anchor=None, ori=0.0, opacity=None, contrast=1.0, depth=0, interpolate=True, lineRGB=False, fillRGB=False, name=None, autoLog=None, autoDraw=False, color=None, colorSpace='rgb')
```

Creates a Circle with a given radius as a special case of a `ShapeStim`

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **edges** (*int*) – Number of edges to use to define the outline of the circle. The greater the number of edges, the ‘rounder’ the circle will appear.
- **radius** (*float*) – Initial radius of the circle in *units*.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **lineWidth** (*float*) – Width of the circle’s outline.
- **lineColor** (*array_like*, *str*, *Color* or *None*) – Color of the circle’s outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **fillColor** (*array_like*, *str*, *Color* or *None*) – Color of the circle’s outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **lineColorSpace** (*str*) – Colorspace to use for the outline and fill. These change how the values passed to *lineColor* and *fillColor* are interpreted. *Deprecated*. Please use *colorSpace* to set both outline and fill colorspace. These arguments may be removed in a future version.
- **fillColorSpace** (*str*) – Colorspace to use for the outline and fill. These change how the values passed to *lineColor* and *fillColor* are interpreted. *Deprecated*. Please use *colorSpace* to set both outline and fill colorspace. These arguments may be removed in a future version.
- **pos** (*array_like*) – Initial position (*x*, *y*) of the circle on-screen relative to the origin located at the center of the window or buffer in *units* (unless changed by specifying *viewPos*). This can be updated after initialization by setting the *pos* property. The default value is *(0.0, 0.0)* which results in no translation.
- **size** (*float or array_like*) – Initial scale factor for adjusting the size of the circle. A single value (*float*) will apply uniform scaling, while an array (*sx*, *sy*) will result in anisotropic scaling in the horizontal (*sx*) and vertical (*sy*) direction. Providing negative values to *size* will cause the shape being mirrored. Scaling can be changed by setting the *size* property after initialization. The default value is *1.0* which results in no scaling.
- **ori** (*float*) – Initial orientation of the circle in degrees about its origin. Positive values will rotate the shape clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.
- **contrast** (*float*) – Contrast level of the shape (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the stimulus when *autoDraw* is enabled.
- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing shape outlines. This produces a smoother (less-pixelated) outline of the shape.
- **lineRGB** (*array_like*, *Color* or *None*) – *Deprecated*. Please use *lineColor* and *fillColor*. These arguments may be removed in a future version.

- **fillRGB** (array_like, *Color* or None) – *Deprecated*. Please use *lineColor* and *fillColor*. These arguments may be removed in a future version.
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the *draw()* method.
- **color** (array_like, str, *Color* or None) – Sets both the initial *lineColor* and *fillColor* of the shape.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.

radius

Radius of the shape. Avoid using *size* for adjusting figure dimensions if radius != 0.5 which will result in undefined behavior.

Type float or int

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

static _calcEquilateralVertices (*edges, radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

_calcPosRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*.

_calcSizeRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*

_getDesiredRGB (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires *self.rgb*, *self.colorSpace* and *self.contrast*

_getPolyAsRendered ()

DEPRECATED. Return a list of vertices as rendered.

_selectWindow (*win*)

Switch drawing to the specified window. Calls the window's *_setCurrent()* method which handles the switch.

_set (*attrib, val, op="", log=None*)

DEPRECATED since 1.80.04 + 1. Use *setAttribute()* and *val2array()* instead.

_updateList ()

The user shouldn't need this method since it gets called after every call to *.set()* Chooses between using and not using shaders each call.

_updateVertices ()

Sets *Stim.verticesPix* and *._borderPix* from *pos*, *size*, *ori*, *flipVert*, *flipHoriz*

autoDraw

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

autoLog

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property backColor

Alternative way of setting fillColor

property backColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property borderColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

closeShape

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, *closeShape=True* is assumed and shouldn't be changed.

color

Set the color of the shape. Sets both *fillColor* and *lineColor* simultaneously if applicable.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains (*x*, *y=None*, *units=None*)

Returns True if a point *x,y* is inside the stimulus' border.

Can accept variety of input options:

- two separate args, *x* and *y*
- one arg (list, tuple or array) containing two vals (*x,y*)

- an object with a `getPos()` method that returns `x,y`, such as a `Mouse`.

Returns `True` if the point is within the area defined either by its `border` attribute (if one defined), or its `vertices` attribute if there is no `.border`. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the `contains` method; the extent of the stimulus is determined purely by the size, position (`pos`), and orientation (`ori`) settings (and by the vertices for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

property `contrast`

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus `opacity` to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

`depth`

DEPRECATED, `depth` is now controlled simply by drawing order.

`draw` (`win=None`, `keepMatrix=False`)

Draw the stimulus in its relevant window.

You must call this method after every `MyWin.flip()` if you want the stimulus to appear on that frame and then update the screen again.

`edges`

Number of edges of the polygon. Floats are rounded to int.

Operations supported.

property `fillColor`

Set the fill color for the shape.

property `fillColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object.

property `fillRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property `flip`

1x2 array for flipping vertices along each axis; set as `True` to flip or `False` to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

property `foreColor`

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other *Color spaces***. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current `colorSpace`. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that `colorSpace='rgb'`:

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use `setColor` if you want to set color and `colorSpace` in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property `foreColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object.

property `foreRGB`

Legacy property for setting the foreground color of a stimulus in RGB, instead use `obj._foreColor.rgb`

Type DEPRECATED

property `interpolate`

If *True* the edge of the line will be anti-aliased.

property `lineColor`

Alternative way of setting `borderColor`.

property lineColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use `obj._borderColor.rgb`

Type DEPRECATED

lineWidth

Width of the line in **pixels**.

Operations supported.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. `visual.TextStim(win)` will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, `shapeContains.py`

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

radius

float, int, tuple, list or 2x1 array Radius of the Polygon (distance from the center to the corners). May be a -2tuple or list to stretch the polygon asymmetrically.

Operations supported.

Usually there's a `setAttribute(value, log=False)` method for each attribute. Use this if you want to disable logging.

setAutoDraw (*value, log=None*)

Sets `autoDraw`. Usually you can use `'stim.attribute = value'` syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True, log=None*)

Usually you can use `'stim.attribute = value'` syntax instead, but use this method if you need to suppress the log message.

setBackRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set `obj._fillColor.rgb`

setBorderColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for `fillColor`, allows suppression of the log message, simultaneous `colorSpace` setting and calls update methods.

setBorderRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set `obj._borderColor.rgb`

setColor (*color, colorSpace=None, operation="", log=None*)

Sets both the line and fill to be the same color.

setContrast (*newContrast, operation="", log=None*)

Usually you can use `'stim.attribute = value'` syntax instead, but use this method if you need to suppress the log message

setDKL (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the `color` attribute

setDepth (*newDepth, operation="", log=None*)

Usually you can use `'stim.attribute = value'` syntax instead, but use this method if you need to suppress the log message

setEdges (*edges, operation="", log=None*)

Usually you can use `'stim.attribute = value'` syntax instead, but use this method if you need to suppress the log message

setFillColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for `fillColor`, allows suppression of the log message, simultaneous `colorSpace` setting and calls update methods.

setFillRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set `obj._fillColor.rgb`

setForeColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for `foreColor`, allows suppression of the log message, simultaneous `colorSpace` setting and calls update methods.

setForeRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set `obj._foreColor.rgb`

setLMS (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOpacity (*newOpacity*, *operation=""*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos (*newPos*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setRadius (*radius*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setSize (*newSize*, *operation=""*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setVertices (*value=None*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.6 CustomMouse

class psychopy.visual.**CustomMouse** (*args, **kwargs)

Class for more control over the mouse, including the pointer graphic and bounding box.

Seems to work with pyglet or pygame. Not completely tested.

Known limitations:

- only norm units are working
- getRel() always returns [0,0]
- mouseMoved() is always False; maybe due to *self.mouse.visible == False* -> held at [0,0]
- no idea if clickReset() works

Author: Jeremy Gray, 2011

Class for customizing the appearance and behavior of the mouse.

Use a custom mouse for extra control over the pointer appearance and function. It's probably slower to render than the regular system mouse. Create your *visual.Window* before creating a CustomMouse.

Parameters

- **win** (required, *visual.Window*) – the window to which this mouse is attached
- **visible** (**True** or **False**) – makes the mouse invisible if necessary
- **newPos** (**None** or [x,y]) – gives the mouse a particular starting position
- **leftLimit** – left edge of a virtual box within which the mouse can move
- **topLimit** – top edge of virtual box
- **rightLimit** – right edge of virtual box
- **bottomLimit** – lower edge of virtual box
- **showLimitBox** (*default is False*) – display the boundary within which the mouse can move.
- **pointer** – The visual display item to use as the pointer; must have *.draw()* and *setPos()* methods. If your item has *.setOpacity()*, you can alter the mouse's opacity.

- **clickOnUp** (*when to count a mouse click as having occurred*) – default is False, record a click when the mouse is first pressed down. True means record a click when the mouse button is released.

9.3.7 DotStim

class psychopy.visual.DotStim(*args, **kwargs)

This stimulus class defines a field of dots with an update rule that determines how they change on every call to the .draw() method.

This single class can be used to generate a wide variety of dot motion types. For a review of possible types and their pros and cons see Scase, Braddick & Raymond (1996). All six possible motions they describe can be generated with appropriate choices of the *signalDots* (which determines whether signal dots are the ‘same’ or ‘different’ on each frame), *noiseDots* (which determines the locations of the noise dots on each frame) and the *dotLife* (which determines for how many frames the dot will continue before being regenerated).

The default settings (as of v1.70.00) is for the noise dots to have identical velocity but random direction and signal dots remain the ‘same’ (once a signal dot, always a signal dot).

For further detail about the different configurations see *Dots (RDK) Component* in the Builder Components section of the documentation.

If further customisation is required, then the DotStim should be subclassed and its `_update_dotsXY` and `_newDotsXY` methods overridden.

The maximum number of dots that can be drawn is limited by system performance.

fieldShape

‘sqr’ or ‘circle’. Defines the envelope used to present the dots. If changed while drawing, dots outside new envelope will be respawned.

Type str

dotSize

Dot size specified in pixels (overridden if *element* is specified). *operations* are supported.

Type float

dotLife

Number of frames each dot lives for (-1=infinite). Dot lives are initiated randomly from a uniform distribution from 0 to dotLife. If changed while drawing, the lives of all dots will be randomly initiated again.

Type int

signalDots

If ‘same’ then the signal and noise dots are constant. If ‘different’ then the choice of which is signal and which is noise gets randomised on each frame. This corresponds to Scase et al’s (1996) categories of RDK.

Type str

noiseDots

Determines the behaviour of the noise dots, taken directly from Scase et al’s (1996) categories. For ‘position’, noise dots take a random position every frame. For ‘direction’ noise dots follow a random, but constant direction. For ‘walk’ noise dots vary their direction every frame, but keep a constant speed.

Type str

element

This can be any object that has a `.draw()` method and a `.setPos([x, y])` method (e.g. a GratingStim, TextStim...!!) DotStim assumes that the element uses pixels as units. None defaults to dots.

Type `object`

fieldPos

Specifying the location of the centre of the stimulus using a *x,y-pair*. See e.g. *ShapeStim* for more documentation/examples on how to set position. *operations* are supported.

Type `array_like`

fieldSize

Specifying the size of the field of dots using a *x,y-pair*. See e.g. *ShapeStim* for more documentation/examples on how to set position. *operations* are supported.

Type `array_like`

coherence

Change the coherence (%) of the DotStim. This will be rounded according to the number of dots in the stimulus.

Type `float`

dir

Direction of the coherent dots in degrees. *operations* are supported.

Type `float`

speed

Speed of the dots (in *units/frame*). *operations* are supported.

Type `float`

Parameters

- **win** (*window.Window*) – Window this stimulus is associated with.
- **units** (*str*) – Units to use.
- **nDots** (*int*) – Number of dots to present in the field.
- **coherence** (*float*) – Proportion of dots which are coherent. This value can be set using the *coherence* property after initialization.
- **fieldPos** (*array_like*) – (x,y) or [x,y] position of the field. This value can be set using the *fieldPos* property after initialization.
- **fieldSize** (*array_like, int or float*) – (x,y) or [x,y] or single value (applied to both dimensions). Sizes can be negative and can extend beyond the window. This value can be set using the *fieldSize* property after initialization.
- **fieldShape** (*str*) – Defines the envelope used to present the dots. If changed while drawing by setting the *fieldShape* property, dots outside new envelope will be respawned., valid values are ‘square’, ‘sqr’ or ‘circle’.
- **dotSize** (*array_like or float*) – Size of the dots. If given an array, the sizes of individual dots will be set. The array must have length *nDots*. If a single value is given, all dots will be set to the same size.
- **dotLife** (*int*) – Lifetime of a dot in frames. Dot lives are initiated randomly from a uniform distribution from 0 to dotLife. If changed while drawing, the lives of all dots will be randomly initiated again. A value of -1 results in dots having an infinite lifetime. This value can be set using the *dotLife* property after initialization.
- **dir** (*float*) – Direction of the coherent dots in degrees. At 0 degrees, coherent dots will move from left to right. Increasing the angle will rotate the direction counter-clockwise. This value can be set using the *dir* property after initialization.

- **speed** (*float*) – Speed of the dots (in *units* per frame). This value can be set using the *speed* property after initialization.
- **rgb** (*array_like*, *optional*) – Color of the dots in form (r, g, b) or [r, g, b]. **Deprecated**, use *color* instead.
- **color** (*array_like or str*) – Color of the dots in form (r, g, b) or [r, g, b].
- **colorSpace** (*str*) – Colorspace to use.
- **opacity** (*float*) – Opacity of the dots from 0.0 to 1.0.
- **contrast** (*float*) – Contrast of the dots 0.0 to 1.0. This value is simply multiplied by the *color* value.
- **depth** (*float*) – **Deprecated**, depth is now controlled simply by drawing order.
- **element** (*object*) – This can be any object that has a `.draw()` method and a `.setPos([x, y])` method (e.g. a `GratingStim`, `TextStim`...!! `DotStim` assumes that the element uses pixels as units. `None` defaults to dots.
- **signalDots** (*str*) – If ‘same’ then the signal and noise dots are constant. If different then the choice of which is signal and which is noise gets randomised on each frame. This corresponds to Scase et al’s (1996) categories of RDK. This value can be set using the *signalDots* property after initialization.
- **noiseDots** (*str*) – Determines the behaviour of the noise dots, taken directly from Scase et al’s (1996) categories. For ‘position’, noise dots take a random position every frame. For ‘direction’ noise dots follow a random, but constant direction. For ‘walk’ noise dots vary their direction every frame, but keep a constant speed. This value can be set using the *noiseDots* property after initialization.
- **name** (*str*, *optional*) – Optional name to use for logging.
- **autoLog** (*bool*) – Enable automatic logging.

9.3.8 ElementArrayStim

class `psychopy.visual.ElementArrayStim(*args, **kwargs)`

This stimulus class defines a field of elements whose behaviour can be independently controlled. Suitable for creating ‘global form’ stimuli or more detailed random dot stimuli.

This stimulus can draw thousands of elements without dropping a frame, but in order to achieve this performance, uses several OpenGL extensions only available on modern graphics cards (supporting OpenGL2.0). See the `ElementArray` demo.

Parameters

win : a *Window* object (required)

units [`None`, ‘height’, ‘norm’, ‘cm’, ‘deg’ or ‘pix’] If `None` then the current units of the *Window* will be used. See *Units for the window and stimuli* for explanation of other options.

nElements : number of elements in the array.

9.3.9 Form

Attributes

<code>Form(win[, name, colorSpace, fillColor, ...])</code>	A class to add Forms to a <i>psychopy.visual.Window</i>
<code>Form.win</code>	The <i>Window</i> object in which the stimulus will be rendered by default.
<code>Form.verticesPix</code>	This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units
<code>Form.values</code>	
<code>Form.updateOpacity()</code>	Placeholder method to update colours when set externally, for example updating the <i>palette</i> attribute of a textbox.
<code>Form.updateColors()</code>	Placeholder method to update colours when set externally, for example updating the <i>palette</i> attribute of a textbox
<code>Form.units</code>	
<code>Form.style</code>	
<code>Form.size</code>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<code>Form.setSize(newSize[, operation, units, log])</code>	Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message
<code>Form.setScrollSpeed(items[, multiplier])</code>	Set scroll speed of Form.
<code>Form.setRGB(color[, operation, log])</code>	DEPRECATED: Legacy setter for foreground RGB, instead set <i>obj._foreColor.rgb</i>
<code>Form.setPos(newPos[, operation, log])</code>	Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.
<code>Form.setOri(newOri[, operation, log])</code>	Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message
<code>Form.setOpacity(newOpacity[, operation, log])</code>	Hard setter for opacity, allows the suppression of log messages and calls the update method
<code>Form.setLineColor(color[, colorSpace, ...])</code>	
<code>Form.setLMS(color[, operation])</code>	DEPRECATED since v1.60.05: Please use the <i>color</i> attribute
<code>Form.setForeColor(color[, colorSpace, ...])</code>	Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.
<code>Form.setFillColor(color[, colorSpace, ...])</code>	Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.
<code>Form.setDepth(newDepth[, operation, log])</code>	Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message
<code>Form.setDKL(color[, operation])</code>	DEPRECATED since v1.60.05: Please use the <i>color</i> attribute

continues on next page

Table 9.6 – continued from previous page

<code>Form.setContrast(newContrast[, operation, log])</code>	Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message
<code>Form.setColor(color[, colorSpace, ...])</code>	
<code>Form.setBorderColor(color[, colorSpace, ...])</code>	Hard setter for <i>fillColor</i> , allows suppression of the log message, simultaneous colorSpace setting and calls update methods.
<code>Form.setBackColor(color[, colorSpace, ...])</code>	
<code>Form.setAutoLog([value, log])</code>	Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.
<code>Form.setAutoDraw(value[, log])</code>	Sets autoDraw for Form and any responseCtrl contained within
<code>Form.pos</code>	The position of the center of the stimulus in the stimulus <i>units</i>
<code>Form.overlaps(polygon)</code>	Returns <i>True</i> if this stimulus intersects another one.
<code>Form.ori</code>	The orientation of the stimulus (in degrees).
<code>Form.opacity</code>	Determines how visible the stimulus is relative to background.
<code>Form.name</code>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<code>Form.lineColor</code>	Alternative way of setting <i>borderColor</i> .
<code>Form.knownStyles</code>	
<code>Form.importItems(items)</code>	Import items from csv or excel sheet and convert to list of dicts.
<code>Form.getData()</code>	Extracts form questions, response ratings and response times from Form items
<code>Form.formComplete()</code>	Deprecated in version 2020.2.
<code>Form.foreColor</code>	Sets both <i>itemColor</i> and <i>responseColor</i> to the same value
<code>Form.fillColor</code>	Color of the form’s background
<code>Form.draw()</code>	Draw all form elements
<code>Form.depth</code>	DEPRECATED, depth is now controlled simply by drawing order.
<code>Form.contrast</code>	A value that is simply multiplied by the color.
<code>Form.contains(x[, y, units])</code>	Returns True if a point x,y is inside the stimulus’ border.
<code>Form.complete</code>	A read-only property to determine if the current form is complete
<code>Form.colorSpace</code>	The name of the color space currently being used
<code>Form.color</code>	Alternative way of setting <i>foreColor</i> .
<code>Form.borderColor</code>	Color of the line around the form
<code>Form.backColor</code>	Alternative way of setting <i>fillColor</i>
<code>Form.autoLog</code>	Whether every change in this stimulus should be auto logged.
<code>Form.autoDraw</code>	Determines whether the stimulus should be automatically drawn on every frame flip.
<code>Form.addDataToExp(exp[, itemsAs])</code>	Gets the current Form data and inserts into an ExperimentHandler object either as rows or as columns

Details

```
class psychopy.visual.Form(win, name='default', colorSpace='rgb', fillColor=None, border-
    Color=None, itemColor='white', responseColor='white', marker-
    Color='red', items=None, font=None, textHeight=0.02, size=0.5, 0.5,
    pos=0, 0, style=None, itemPadding=0.05, units='height', random-
    ize=False, autoLog=True, color=None, foreColor=None)
```

A class to add Forms to a *psychopy.visual.Window*

The Form allows PsychoPy to be used as a questionnaire tool, where participants can be presented with a series of questions requiring responses. Form items, defined as questions and response pairs, are presented simultaneously onscreen with a scrollable viewing window.

Example

```
survey = Form(win, items=[{}], size=(1.0, 0.7), pos=(0.0, 0.0))
```

Parameters

- **win** (*psychopy.visual.Window*) – The window object to present the form.
- **items** (*List of dicts or csv or xlsx file*) –
a list of dicts or csv file should have the following key, value pairs / column headers:
 "index": The item index as a number "itemText": item question string, "itemWidth":
 fraction of the form width 0:1 "type": type of rating e.g., 'radio', 'rating', 'slider'
 "responseWidth": fraction of the form width 0:1, "options": list of tick labels for options,
 "layout": Response object layout e.g., 'horiz' or 'vert'
- **textHeight** (*float*) – Text height.
- **size** (*tuple, list*) – Size of form on screen.
- **pos** (*tuple, list*) – Position of form on screen.
- **itemPadding** (*float*) – Space or padding between form items.
- **units** (*str*) – units for stimuli - Currently, Form class only operates with 'height' units.
- **randomize** (*bool*) – Randomize order of Form elements

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`

`_createItemCtrls()`

Define layout of form

`_drawCtrls()`

Draw elements on form within border range.

Parameters **items** (*List*) – List of TextStim or Slider item from survey

`_drawDecorations()`

Draw decorations on form.

`_drawExternalDecorations()`

Draw decorations outside the aperture

`_getDesiredRGB(rgb, colorSpace, contrast)`

Convert color to RGB while adding contrast. Requires `self.rgb`, `self.colorSpace` and `self.contrast`

`_getItemHeight(item, ctrl=None)`

Returns the full height of the item to be inserted in the form

`_getItemRenderedWidth(size)`

Returns text width for item text based on `itemWidth` and Form width.

Parameters `size` (*float, int*) – The question width

Returns Wrap width for question text

Return type `float`

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_getScrollOffset()`

Calculate offset position of items in relation to `markerPos`. Offset is a proportion of *height - height*, meaning the max offset (when `scrollbar.markerPos` is 1) is enough to take the bottom element to the bottom of the border.

Returns Offset position of items proportionate to scroll bar

Return type `float`

`_inRange(item)`

Check whether item position falls within border area

Parameters `item` (*TextStim, Slider object*) – TextStim or Slider item from survey

Returns Returns True if item position falls within border area

Return type `bool`

`_layoutY()`

This needs to be done when editable textboxes change their size because everything below them needs to move too

`_makeSlider(item)`

Creates Slider object for Form class

Parameters

- `item` (*dict*) – The dict entry for a single item
- `pos` (*tuple*) – position of response object

Returns

- `psychopy.visual.slider.Slider` – The Slider object for response
- `respHeight` – The height of the response object as type float

`_makeTextBox(item)`

Creates TextBox object for Form class

NOTE: The TextBox 2 in work in progress, and has not been added to Form class yet. :param item: The dict entry for a single item :type item: dict :param pos: position of response object :type pos: tuple

Returns

- `psychopy.visual.TextBox` – The TextBox object for response

- *respHeight* – The height of the response object as type float

`_selectWindow` (*win*)

Switch drawing to the specified window. Calls the window's `_setCurrent()` method which handles the switch.

`_set` (*attrib, val, op="", log=None*)

DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

`_setAperture` ()

Blocks text beyond border using Aperture

Returns The aperture setting viewable area for forms

Return type *psychopy.visual.Aperture*

`_setBorder` ()

Creates border using Rect

Returns The border for the survey

Return type *psychopy.visual.Rect*

`_setDecorations` ()

Sets Form decorations i.e., Border and scrollbar

`_setQuestion` (*item*)

Creates TextStim object containing question

Parameters *item* (*dict*) – The dict entry for a single item

Returns

- *psychopy.visual.text.TextStim* – The textstim object with the question string
- *questionHeight* – The height of the question bounding box as type float
- *questionWidth* – The width of the question bounding box as type float

`_setResponse` (*item*)

Makes calls to methods which make Slider or TextBox response objects for Form

Parameters

- *item* (*dict*) – The dict entry for a single item
- *question* (*TextStim*) – The question text object

Returns

- *psychopy.visual.slider.Slider* – The Slider object for response
- *psychopy.visual.TextBox* – The TextBox object for response
- *respHeight* – The height of the response object as type float

`_setScrollBar` ()

Creates Slider object for scrollbar

Returns The Slider object for scroll bar

Return type *psychopy.visual.slider.Slider*

`_updateList` ()

The user shouldn't need this method since it gets called after every call to `.set()` Chooses between using and not using shaders each call.

`_updateVertices()`

Sets `Stim.verticesPix` and `._borderPix` from `pos`, `size`, `ori`, `flipVert`, `flipHoriz`

`addDataToExp(exp, itemsAs='rows')`

Gets the current Form data and inserts into an `ExperimentHandler` object either as rows or as columns

Parameters

- `exp` (`ExperimentHandler`) –
- `itemsAs` (`'rows'`, `'cols'` (or `'columns'`)) –

property `anchor`

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property `backColor`

Alternative way of setting `fillColor`

property `backColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object.

property `backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property `borderColor`

Color of the line around the form

property `borderColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object

property `borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use `obj._borderColor.rgb`

Type DEPRECATED

property `color`

Alternative way of setting `foreColor`.

property `colorSpace`

The name of the color space currently being used

Value should be: a string or `None`

For strings and hex values this is not needed. If `None` the default `colorSpace` for the stimulus is used (defined during initialisation).

Please note that changing `colorSpace` does not change stimulus parameters. Thus you usually want to specify `colorSpace` before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property complete

A read-only property to determine if the current form is complete

contains (*x, y=None, units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a `getPos()` method that returns x,y, such** as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (*pos*), and orientation (*ori*) settings (and by the vertices for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw()

Draw all form elements

property fillColor

Color of the form's background

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property foreColor

Sets both *itemColor* and *responseColor* to the same value

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

formComplete ()

Deprecated in version 2020.2. Please use the Form.complete property

getData ()

Extracts form questions, response ratings and response times from Form items

Returns A copy of the data as a list of dicts

Return type list

property height

importItems (items)

Import items from csv or excel sheet and convert to list of dicts. Will also accept a list of dicts.

Note, for csv and excel files, 'options' must contain comma separated values, e.g., one, two, three. No parenthesis, or quotation marks required.

Parameters *items* (*Excel or CSV file, list of dicts*) – Items used to populate the Form

Returns A list of dicts, where each list entry is a dict containing all fields for a single Form item

Return type List of dicts

property itemColor

Color of the text on form items

knownStyles = {'dark': {'borderColor': None, 'fillColor': [-0.19, -0.19, -0.14], 'f

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

property markerColor

Color of the marker on any sliders in this form

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (polygon)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

reset ()

Clear all responses and set all items to their initial values.

property responseColor

Color of the lines and text on form responses

property scrollbarWidth

Width of the scrollbar for this Form, in the spatial units of this Form. Can also be set as a *layout.Vector* object.

setAnchor (*value*, *log=None*)

setAutoDraw (*value*, *log=None*)

Sets autoDraw for Form and any responseCtrl contained within

setAutoLog (*value=True*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setBackColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setBackRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setBorderRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setContrast (*newContrast*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setDKL (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setFillColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setFillRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *foreColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setForeRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setLineRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOpacity (*newOpacity*, *operation=""*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos (*newPos*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setScrollSpeed (*items*, *multiplier=2*)

Set scroll speed of Form. Higher multiplier gives smoother, but slower scroll.

Parameters

- **items** (*list of dicts*) – Items used to populate the form
- **multiplier** (*int (default=2)*) – Number used to calculate scroll speed

Returns Scroll speed, calculated using N items by multiplier

Return type `int`

setSize (*newSize*, *operation=""*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or *None* (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

property style

property units

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property values

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.10 GratingStim

Attributes

<i>GratingStim(win[, tex, mask, units, anchor, ...])</i>	Stimulus object for drawing arbitrary bitmaps that can repeat (cycle) in either dimension.
<i>GratingStim.win</i>	The <i>Window</i> object in which the stimulus will be rendered by default.
<i>GratingStim.tex</i>	Texture to used on the stimulus as a grating (aka carrier)
<i>GratingStim.mask</i>	The alpha mask (forming the shape of the image).
<i>GratingStim.units</i>	
<i>GratingStim.sf</i>	Spatial frequency of the grating texture
<i>GratingStim.pos</i>	The position of the center of the stimulus in the stimulus <i>units</i>
<i>GratingStim.ori</i>	The orientation of the stimulus (in degrees).
<i>GratingStim.size</i>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<i>GratingStim.contrast</i>	A value that is simply multiplied by the color.
<i>GratingStim.color</i>	Alternative way of setting <i>foreColor</i> .
<i>GratingStim.colorSpace</i>	The name of the color space currently being used
<i>GratingStim.opacity</i>	Determines how visible the stimulus is relative to background.
<i>GratingStim.interpolate</i>	Whether to interpolate (linearly) the texture in the stimulus.
<i>GratingStim.texRes</i>	Power-of-two int.

continues on next page

Table 9.7 – continued from previous page

<code>GratingStim.name</code>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<code>GratingStim.autoLog</code>	Whether every change in this stimulus should be auto logged.
<code>GratingStim.draw([win])</code>	Draw the stimulus in its relevant window.
<code>GratingStim.autoDraw</code>	Determines whether the stimulus should be automatically drawn on every frame flip.

Details

```
class psychopy.visual.GratingStim(win, tex='sin', mask='none', units=None, anchor='center',
    pos=0.0, 0.0, size=None, sf=None, ori=0.0, phase=0.0, 0.0,
    texRes=128, rgb=None, dkl=None, lms=None, color=1.0,
    1.0, 1.0, colorSpace='rgb', contrast=1.0, opacity=None,
    depth=0, rgbPedestal=0.0, 0.0, 0.0, interpolate=False,
    blendmode='avg', name=None, autoLog=None, auto-
    Draw=False, maskParams=None)
```

Stimulus object for drawing arbitrary bitmaps that can repeat (cycle) in either dimension.

One of the main stimuli for PsychoPy.

Formally GratingStim is just a texture behind an optional transparency mask (an ‘alpha mask’). Both the texture and mask can be arbitrary bitmaps and their combination allows an enormous variety of stimuli to be drawn in realtime.

Examples:

```
myGrat = GratingStim(tex='sin', mask='circle') # circular grating
myGabor = GratingStim(tex='sin', mask='gauss') # gives a 'Gabor'
```

A GratingStim can be rotated scaled and shifted in position, its texture can be drifted in X and/or Y and it can have a spatial frequency in X and/or Y (for an image file that simply draws multiple copies in the patch).

Also since transparency can be controlled two GratingStims can combine e.g. to form a plaid.

Using GratingStim with images from disk (jpg, tif, png, ...)

Ideally texture images to be rendered should be square with ‘power-of-2’ dimensions e.g. 16 x 16, 128 x 128. Any image that is not will be upsampled (with linear interpolation) to the nearest such texture by PsychoPy. The size of the stimulus should be specified in the normal way using the appropriate units (deg, pix, cm, ...). Be sure to get the aspect ratio the same as the image (if you don’t want it stretched!).

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use `obj._foreColor.rgb`

Type DEPRECATED

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`

`_createTexture`(tex, id, pixFormat, stim, res=128, maskParams=None, forcePOW2=True, dataType=None, wrapping=True)

Create a new OpenGL 2D image texture.

Parameters

- **tex** (*Any*) – Texture data. Value can be anything that resembles image data.
- **id** (int or GLint) – Texture ID.
- **pixFormat** (GLenum or int) – Pixel format to use, values can be *GL_ALPHA* or *GL_RGB*.
- **stim** (*Any*) – Stimulus object using the texture.
- **res** (*int*) – The resolution of the texture (unless a bitmap image is used).
- **maskParams** (*dict or None*) – Additional parameters to configure the mask used with this texture.
- **forcePOW2** (*bool*) – Force the texture to be stored in a square memory area. For grating stimuli (anything that needs multiple cycles) *forcePOW2* should be set to be *True*. Otherwise the wrapping of the texture will not work.
- **dataType** (class:~pyglet.gl.*GLenum*, int or None) – None, *GL_UNSIGNED_BYTE*, *GL_FLOAT*. Only affects image files (numpy arrays will be float).
- **wrapping** (*bool*) – Enable wrapping of the texture. A texture will be set to repeat (or tile).

__getDesiredRGB (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires self.rgb, self.colorSpace and self.contrast

__getPolyAsRendered ()

DEPRECATED. Return a list of vertices as rendered.

__selectWindow (*win*)

Switch drawing to the specified window. Calls the window's `__setCurrent()` method which handles the switch.

__set (*attrib, val, op="", log=None*)

DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

__updateList ()

The user shouldn't need this method since it gets called after every call to `.set()` Chooses between using and not using shaders each call.

__updateListShaders ()

The user shouldn't need this method since it gets called after every call to `.set()` Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the `.set()` command

__updateVertices ()

Sets `Stim.verticesPix` and `._borderPix` from `pos`, `size`, `ori`, `flipVert`, `flipHoriz`

property anchor

autoDraw

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

autoLog

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property backColor

Alternative way of setting fillColor

property backColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

blendmode

The OpenGL mode in which the stimulus is draw

Can be 'avg' or 'add'. Average (avg) places the new stimulus over the old one with a transparency given by its opacity. Opaque stimuli will hide other stimuli transparent stimuli won't. Add performs the arithmetic sum of the new stimulus and the ones already present.

property borderColor

property borderColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

clearTextures ()

Clear all textures associated with the stimulus.

As of v1.61.00 this is called automatically during garbage collection of your stimulus, so doesn't need calling explicitly by the user.

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains (x, y=None, units=None)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y

- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a `getPos()` method that returns x,y, such** as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (*pos*), and orientation (*ori*) settings (and by the vertices for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

property `contrast`

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

`depth`

DEPRECATED, `depth` is now controlled simply by drawing order.

`draw` (*win=None*)

Draw the stimulus in its relevant window. You must call this method after every `MyWin.flip()` if you want the stimulus to appear on that frame and then update the screen again.

property `fillColor`

Set the fill color for the shape.

property `fillColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object.

property `fillRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property `flip`

1x2 array for flipping vertices along each axis; set as `True` to flip or `False` to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

property `flipHoriz`

property `flipVert`

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

property height

interpolate

Whether to interpolate (linearly) the texture in the stimulus.

If set to False then nearest neighbour will be used when needed, otherwise some form of interpolation will be used.

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

mask

The alpha mask (forming the shape of the image).

This can be one of various options:

- ‘circle’, ‘gauss’, ‘raisedCos’, ‘cross’
- **None** (resets to default)
- the name of an image file (most formats supported)
- a numpy array (1xN or NxN) ranging -1:1

maskParams

Various types of input. Default to *None*.

This is used to pass additional parameters to the mask if those are needed.

- **For ‘gauss’ mask, pass dict {‘sd’: 5} to control** standard deviation.
- **For the ‘raisedCos’ mask, pass a dict: {‘fringeWidth’:0.2}**, where ‘fringeWidth’ is a parameter (float, 0-1), determining the proportion of the patch that will be blurred by the raised cosine edge.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. `visual.TextStim(win)` will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, `shapeContains.py`

phase

Phase of the stimulus in each dimension of the texture.

Should be an *x,y-pair* or *scalar*

NB phase has modulus 1 (rather than 360 or $2*\pi$) This is a little unconventional but has the nice effect that setting `phase=t*n` drifts a stimulus at n Hz

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAnchor (*value*, *log=None*)

setAutoDraw (*value*, *log=None*)

Sets `autoDraw`. Usually you can use `'stim.attribute = value'` syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True*, *log=None*)

Usually you can use `'stim.attribute = value'` syntax instead, but use this method if you need to suppress the log message.

setBackColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setBackRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set `obj._fillColor.rgb`

setBlendmode (*value*, *log=None*)

DEPRECATED. Use `'stim.parameter = value'` syntax instead

setBorderColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for `fillColor`, allows suppression of the log message, simultaneous `colorSpace` setting and calls update methods.

setBorderRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set `obj._borderColor.rgb`

setColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setContrast (*newContrast*, *operation=""*, *log=None*)

Usually you can use `'stim.attribute = value'` syntax instead, but use this method if you need to suppress the log message

setDKL (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setFillColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setLineRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setMask (*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setOpacity (*newOpacity*, *operation=""*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPhase (*value*, *operation=""*, *log=None*)

DEPRECATED. Use 'stim.parameter = value' syntax instead

setPos (*newPos*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSF (*value*, *operation=""*, *log=None*)

DEPRECATED. Use 'stim.parameter = value' syntax instead

setSize (*newSize*, *operation=""*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setTex (*value*, *log=None*)

DEPRECATED. Use 'stim.parameter = value' syntax instead

sf

Spatial frequency of the grating texture

Should be a *x,y-pair* or *scalar* or None. If *units* == 'deg' or 'cm' units are in cycles per deg or cm as appropriate. If *units* == 'norm' then sf units are in cycles per stimulus (and so SF scales with stimulus size). If texture is an image loaded from a file then sf=None defaults to 1/stimSize to give one cycle of the image.

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

tex

Texture to used on the stimulus as a grating (aka carrier)

This can be one of various options:

- 'sin', 'sqr', 'saw', 'tri', None (resets to default)
- the name of an image file (most formats supported)
- a numpy array (1xN or NxN) ranging -1:1

If specifying your own texture using an image or numpy array you should ensure that the image has square power-of-two dimensnions (e.g. 256 x 256). If not then PsychoPy will upsample your stimulus to the next larger power of two.

texRes

Power-of-two int. Sets the resolution of the mask and texture. texRes is overridden if an array or image is provided as mask.

Operations supported.

property units

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.11 Helper functions

`psychoPy.visual.helpers.pointInPolygon(x, y, poly)`

Determine if a point is inside a polygon; returns True if inside.

(*x*, *y*) is the point to test. *poly* is a list of 3 or more vertices as (*x*,*y*) pairs. If given an object, such as a *ShapeStim*, will try to use its vertices and position as the polygon.

Same as the `.contains()` method elsewhere.

`psychoPy.visual.helpers.polygonsOverlap(poly1, poly2)`

Determine if two polygons intersect; can fail for very pointy polygons.

Accepts two polygons, as lists of vertices (*x*,*y*) pairs. If given an object with with (vertices + pos), will try to use that as the polygon.

Checks if any vertex of one polygon is inside the other polygon. Same as the `.overlaps()` method elsewhere.

Notes

We implement special handling for the *Line* stimulus as it is not a proper polygon. We do not check for class instances because this would require importing of *visual.Line*, creating a circular import. Instead, we assume that a “polygon” with only two vertices is meant to specify a line. Pixels between the endpoints get interpolated before testing for overlap.

`psychoPy.visual.helpers.groupFlipVert(flipList, yReflect=0)`

Reverses the vertical mirroring of all items in list `flipList`.

Reverses the `.flipVert` status, vertical (*y*) positions, and angular rotation (`.ori`). Flipping preserves the relations among the group’s visual elements. The parameter `yReflect` is the *y*-value of an imaginary horizontal line around which to reflect the items; default = 0 (screen center).

Typical usage is to call once prior to any display; call again to un-flip. Can be called with a list of all stim to be presented in a given routine.

Will flip a) all `psychoPy.visual.xyzStim` that have a `setFlipVert` method, b) the *y* values of `.vertices`, and c) items in *n* x 2 lists that are mutable (i.e., list, `np.array`, no tuples): `[[x1, y1], [x2, y2], ...]`

9.3.12 ImageStim

As of version 1.79.00 *some* of the properties for this stimulus can be set using the syntax:

```
stim.pos = newPos
```

others need to be set with the older syntax:

```
stim.setImage(newImage)
```

Attributes

<code>ImageStim(win[, image, mask, units, pos, ...])</code>	Display an image on a <i>psychopy.visual.Window</i>
<code>ImageStim.win</code>	The <i>Window</i> object in which the stimulus will be rendered by default.
<code>ImageStim.setImage(value[, log])</code>	Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.
<code>ImageStim.setMask(value[, log])</code>	Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.
<code>ImageStim.units</code>	
<code>ImageStim.pos</code>	The position of the center of the stimulus in the stimulus <i>units</i>
<code>ImageStim.ori</code>	The orientation of the stimulus (in degrees).
<code>ImageStim.size</code>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<code>ImageStim.contrast</code>	A value that is simply multiplied by the color.
<code>ImageStim.color</code>	Alternative way of setting <i>foreColor</i> .
<code>ImageStim.colorSpace</code>	The name of the color space currently being used
<code>ImageStim.opacity</code>	Determines how visible the stimulus is relative to background.
<code>ImageStim.interpolate</code>	Whether to interpolate (linearly) the texture in the stimulus.
<code>ImageStim.contains(x[, y, units])</code>	Returns True if a point x,y is inside the stimulus' border.
<code>ImageStim.overlaps(polygon)</code>	Returns <i>True</i> if this stimulus intersects another one.
<code>ImageStim.name</code>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<code>ImageStim.autoLog</code>	Whether every change in this stimulus should be auto logged.
<code>ImageStim.draw([win])</code>	Draw.
<code>ImageStim.autoDraw</code>	Determines whether the stimulus should be automatically drawn on every frame flip.
<code>ImageStim.clearTextures()</code>	Clear all textures associated with the stimulus.

Details

class `psychoPy.visual.ImageStim`(*win*, *image=None*, *mask=None*, *units=""*, *pos=0.0, 0.0*, *size=None*, *anchor='center'*, *ori=0.0*, *color=1.0, 1.0, 1.0*, *colorSpace='rgb'*, *contrast=1.0*, *opacity=None*, *depth=0*, *interpolate=False*, *flipHoriz=False*, *flipVert=False*, *texRes=128*, *name=None*, *autoLog=None*, *maskParams=None*)

Display an image on a `psychoPy.visual.Window`

property **RGB**

Legacy property for setting the foreground color of a stimulus in RGB, instead use `obj._foreColor.rgb`

Type DEPRECATED

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`

`_createTexture`(*tex*, *id*, *pixFormat*, *stim*, *res=128*, *maskParams=None*, *forcePOW2=True*, *dataType=None*, *wrapping=True*)

Create a new OpenGL 2D image texture.

Parameters

- **tex** (*Any*) – Texture data. Value can be anything that resembles image data.
- **id** (int or `GLint`) – Texture ID.
- **pixFormat** (`GLenum` or int) – Pixel format to use, values can be `GL_ALPHA` or `GL_RGB`.
- **stim** (*Any*) – Stimulus object using the texture.
- **res** (*int*) – The resolution of the texture (unless a bitmap image is used).
- **maskParams** (*dict or None*) – Additional parameters to configure the mask used with this texture.
- **forcePOW2** (*bool*) – Force the texture to be stored in a square memory area. For grating stimuli (anything that needs multiple cycles) *forcePOW2* should be set to be *True*. Otherwise the wrapping of the texture will not work.
- **dataType** (class:~`pyglet.gl.GGLenum`, int or None) – None, `GL_UNSIGNED_BYTE`, `GL_FLOAT`. Only affects image files (numpy arrays will be float).
- **wrapping** (*bool*) – Enable wrapping of the texture. A texture will be set to repeat (or tile).

`_getDesiredRGB`(*rgb*, *colorSpace*, *contrast*)

Convert color to RGB while adding contrast. Requires `self.rgb`, `self.colorSpace` and `self.contrast`

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_movieFrameToTexture`(*movieSrc*)

Convert a movie frame to a texture and use it.

This method is used internally to copy pixel data from a camera object into a texture. This enables the *ImageStim* to be used as a ‘viewfinder’ of sorts for the camera to view a live video stream on a window.

Parameters **movieSrc** (~`psychoPy.hardware.camera.Camera`) – Movie source object.

`_selectWindow` (*win*)
 Switch drawing to the specified window. Calls the window's `_setCurrent()` method which handles the switch.

`_set` (*attrib, val, op="", log=None*)
 DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

`_updateList` ()
 The user shouldn't need this method since it gets called after every call to `.set()` Chooses between using and not using shaders each call.

`_updateListShaders` ()
 The user shouldn't need this method since it gets called after every call to `.set()` Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the `.set()` command

`_updateVertices` ()
 Sets `Stim.verticesPix` and `._borderPix` from `pos, size, ori, flipVert, flipHoriz`

property anchor

autoDraw
 Determines whether the stimulus should be automatically drawn on every frame flip.
 Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

autoLog
 Whether every change in this stimulus should be auto logged.
 Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property backColor
 Alternative way of setting `fillColor`

property backColorSpace
 Deprecated, please use `colorSpace` to set color space for the entire object.

property backRGB
 Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`
 Type DEPRECATED

property borderColor

property borderColorSpace
 Deprecated, please use `colorSpace` to set color space for the entire object

property borderRGB
 Legacy property for setting the border color of a stimulus in RGB, instead use `obj._borderColor.rgb`
 Type DEPRECATED

clearTextures ()
 Clear all textures associated with the stimulus.
 As of v1.61.00 this is called automatically during garbage collection of your stimulus, so doesn't need calling explicitly by the user.

property color
 Alternative way of setting `foreColor`.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains (*x, y=None, units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such** as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw (*win=None*)

Draw.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
                 # space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

property height

image

The image file to be presented (most formats supported).

This can be a path-like object to an image file, or a numpy array of shape [H, W, C] where C are channels. The third dim will usually have length 1 (defining an intensity-only image), 3 (defining an RGB image) or 4 (defining an RGBA image).

If passing a numpy array to the image attribute, the size attribute of ImageStim must be set explicitly.

interpolate

Whether to interpolate (linearly) the texture in the stimulus.

If set to False then nearest neighbour will be used when needed, otherwise some form of interpolation will be used.

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

mask

The alpha mask that can be used to control the outer shape of the stimulus

- **None**, 'circle', 'gauss', 'raisedCos'
- or the name of an image file (most formats supported)
- or a numpy array (1xN or NxN) ranging -1:1

maskParams

Various types of input. Default to *None*.

This is used to pass additional parameters to the mask if those are needed.

- For ‘gauss’ mask, pass dict {‘sd’: 5} to control standard deviation.
- For the ‘raisedCos’ mask, pass a dict: {‘fringeWidth’:0.2}, where ‘fringeWidth’ is a parameter (float, 0-1), determining the proportion of the patch that will be blurred by the raised cosine edge.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAnchor (*value, log=None*)

setAutoDraw (*value, log=None*)

Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True, log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setBackColor (*color, colorSpace=None, operation="", log=None*)

setBackRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setBorderRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor (*color, colorSpace=None, operation="", log=None*)

setContrast (*newContrast, operation="", log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setDKL (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth, operation="", log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setFillColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setFillRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *foreColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setForeRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setImage (*value, log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setLMS (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor (*color, colorSpace=None, operation="", log=None*)

setLineRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setMask (*value, log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setOpacity (*newOpacity, operation="", log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos (*newPos*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize (*newSize*, *operation=""*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

texRes

Power-of-two int. Sets the resolution of the mask and texture. texRes is overridden if an array or image is provided as mask.

Operations supported.

property units

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.13 LightSource

Attributes

<i>LightSource</i> (win[, pos, diffuseColor, ...])	Class for representing a light source in a scene.
--	---

Details

class psychopy.visual.**LightSource** (win, pos=0.0, 0.0, 0.0, diffuseColor=1.0, 1.0, 1.0, specularColor=1.0, 1.0, 1.0, ambientColor=0.0, 0.0, 0.0, colorSpace='rgb', lightType='point', attenuation=1, 0, 0)

Class for representing a light source in a scene.

Only point and directional lighting is supported by this object for now. The ambient color of the light source contributes to the scene ambient color defined by *ambientLight*.

Warning: This class is experimental and may result in undefined behavior.

Parameters

- **win** (~*psychopy.visual.Window*) – Window associated with this light source.
- **pos** (*array_like*) – Position of the light source (x, y, z, w). If *w=1.0* the light will be a point source and *x*, *y*, and *z* is the position in the scene. If *w=0.0*, the light source will be directional and *x*, *y*, and *z* will define the vector pointing to the direction the light source is coming from. For instance, a vector of (0, 1, 0, 0) will indicate that a light source is coming from above.
- **diffuseColor** (*array_like*) – Diffuse light color.
- **specularColor** (*array_like*) – Specular light color.
- **ambientColor** (*array_like*) – Ambient light color.
- **colorSpace** (*str*) – Colorspace for *diffuse*, *specular*, and *ambient* colors.
- **attenuation** (*array_like*) – Values for the constant, linear, and quadratic terms of the lighting attenuation formula. Default is (1, 0, 0) which results in no attenuation.

- property ambientColor**
Ambient color of the material.
- property ambientRGB**
Diffuse color of the material.
- property attenuation**
Values for the constant, linear, and quadratic terms of the lighting attenuation formula.
- property diffuseColor**
Diffuse color of the material.
- property diffuseRGB**
Diffuse color of the material.
- property lightType**
Type of light source, can be 'point' or 'directional'.
- property pos**
Position of the light source in the scene in scene units.
- property specularColor**
Specular color of the material.
- property specularRGB**
Diffuse color of the material.

9.3.14 psychopy.visual.Line

Stimulus class for drawing lines.

Overview

<i>Line</i> (win[, start, end, units, lineWidth, ...])	Creates a Line between two points.
<i>Line.start</i>	tuple, list or 2x1 array.
<i>Line.end</i>	tuple, list or 2x1 array
<i>Line.units</i>	
<i>Line.lineWidth</i>	Width of the line in pixels .
<i>Line.lineColor</i>	Alternative way of setting <i>borderColor</i> .
<i>Line.lineColorSpace</i>	Deprecated, please use <i>colorSpace</i> to set color space for the entire object
<i>Line.fillColor</i>	Set the fill color for the shape.
<i>Line.fillColorSpace</i>	Deprecated, please use <i>colorSpace</i> to set color space for the entire object.
<i>Line.pos</i>	The position of the center of the stimulus in the stimulus <i>units</i>
<i>Line.size</i>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<i>Line.ori</i>	The orientation of the stimulus (in degrees).
<i>Line.opacity</i>	Determines how visible the stimulus is relative to background.
<i>Line.contrast</i>	A value that is simply multiplied by the color.
<i>Line.depth</i>	DEPRECATED, depth is now controlled simply by drawing order.

continues on next page

Table 9.10 – continued from previous page

<code>Line.interpolate</code>	If <i>True</i> the edge of the line will be anti-aliased.
<code>Line.lineRGB</code>	Legacy property for setting the border color of a stimulus in RGB, instead use <code>obj._borderColor.rgb</code>
<code>Line.fillRGB</code>	Legacy property for setting the fill color of a stimulus in RGB, instead use <code>obj._fillColor.rgb</code>
<code>Line.name</code>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<code>Line.autoLog</code>	Whether every change in this stimulus should be auto logged.
<code>Line.autoDraw</code>	Determines whether the stimulus should be automatically drawn on every frame flip.
<code>Line.color</code>	Set the color of the shape.
<code>Line.colorSpace</code>	The name of the color space currently being used

Details

```
class psychopy.visual.line.Line(win, start=- 0.5, - 0.5, end=0.5, 0.5, units=None,
                                lineWidth=1.5, lineColor='white', fillColor=None, lineColorSpace=None, pos=0, 0, size=1.0, anchor='center', ori=0.0,
                                opacity=None, contrast=1.0, depth=0, interpolate=True, lineRGB=False, fillRGB=False, name=None, autoLog=None,
                                autoDraw=False, color=None, colorSpace='rgb')
```

Creates a Line between two points.

Line accepts all input parameters, that *ShapeStim* accepts, except for *vertices*, *closeShape* and *fillColor*.

(New in version 1.72.00)

Parameters

- **win** (*Window*) – Window this line is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **start** (*array_like*) – Coordinate (*x*, *y*) of the starting point of the line.
- **end** (*array_like*) – Coordinate (*x*, *y*) of the end-point of the line.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **lineWidth** (*float*) – Width of the line.
- **lineColor** (*array_like*, *str*, *Color* or *None*) – Color of the line. If *None*, a fully transparent color is used which makes the line invisible. *Deprecated* use *color* instead.
- **lineColorSpace** (*str* or *None*) – Colorspace to use for the line. These change how the values passed to *lineColor* are interpreted. *Deprecated*. Please use *colorSpace* to set the line colorspace. This arguments may be removed in a future version.
- **pos** (*array_like*) – Initial translation (*x*, *y*) of the line on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is (*0.0*, *0.0*) which results in no translation.
- **size** (*float* or *array_like*) – Initial scale factor for adjusting the size of the line. A single value (*float*) will apply uniform scaling, while an array (*sx*, *sy*) will result in anisotropic scaling in the horizontal (*sx*) and vertical (*sy*) direction. Providing negative

values to *size* will cause the line to be mirrored. Scaling can be changed by setting the *size* property after initialization. The default value is *1.0* which results in no scaling.

- **ori** (*float*) – Initial orientation of the line in degrees about its origin. Positive values will rotate the line clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the line. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.
- **contrast** (*float*) – Contrast level of the line (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the stimulus when *autoDraw* is enabled.
- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing lines. This produces a smoother (less-pixelated) line.
- **lineRGB** (array_like, *Color* or None) – *Deprecated*. Please use *color* instead. This argument may be removed in a future version.
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the *draw()* method.
- **color** (array_like, str, *Color* or None) – Sets both the initial *lineColor* and *fillColor* of the shape.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.

Notes

The *contains* method always return *False* because a line is not a proper (2D) polygon.

start, end

Coordinates (*x, y*) for the start- and end-point of the line.

Type array_like

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

static _calcEquilateralVertices (*edges, radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

_calcPosRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*.

_calcSizeRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*

_getDesiredRGB (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires *self.rgb*, *self.colorSpace* and *self.contrast*

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_selectWindow(win)`

Switch drawing to the specified window. Calls the window's `_setCurrent()` method which handles the switch.

`_set(attrib, val, op="", log=None)`

DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

`_tessellate(newVertices)`

Set the `.vertices` and `.border` to new values, invoking tessellation.

`_updateList()`

The user shouldn't need this method since it gets called after every call to `.set()` Chooses between using and not using shaders each call.

`_updateVertices()`

Sets `Stim.verticesPix` and `._borderPix` from `pos`, `size`, `ori`, `flipVert`, `flipHoriz`

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property `backColor`

Alternative way of setting `fillColor`

property `backColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object.

property `backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property `borderColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object

property `borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use `obj._borderColor.rgb`

Type DEPRECATED

`closeShape`

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, `closeShape=True` is assumed and shouldn't be changed.

property `color`

Set the color of the shape. Sets both `fillColor` and `lineColor` simultaneously if applicable.

property `colorSpace`

The name of the color space currently being used

Value should be: a string or `None`

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains (*args, **kwargs)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such** as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (pos), and orientation (ori) settings (and by the vertices for shape stimuli).

See Coder demos: shapeContains.py See Coder demos: shapeContains.py

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw (*win=None, keepMatrix=False*)

Draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

end

tuple, list or 2x1 array

Specifies the position of the end of the line. *Operations* supported.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current *colorSpace*. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that *colorSpace='rgb'*:

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
                 # space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use `setColor` if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property `foreColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object.

property `foreRGB`

Legacy property for setting the foreground color of a stimulus in RGB, instead use `obj._foreColor.rgb`

Type DEPRECATED

property `interpolate`

If `True` the edge of the line will be anti-aliased.

property `lineColor`

Alternative way of setting `borderColor`.

property `lineColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object

property `lineRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use `obj._borderColor.rgb`

Type DEPRECATED

property `lineWidth`

Width of the line in **pixels**.

Operations supported.

property `name`

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. `visual.TextStim(win)` will be called “unnamed TextStim” in the logs.

property `opacity`

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

property `ori`

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

property `overlaps` (*polygon*)

Returns `True` if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, `shapeContains.py`

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAutoDraw (*value*, *log=None*)

Sets `autoDraw`. Usually you can use ‘`stim.attribute = value`’ syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True*, *log=None*)

Usually you can use ‘`stim.attribute = value`’ syntax instead, but use this method if you need to suppress the log message.

setBackRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set `obj._fillColor.rgb`

setBorderColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for `fillColor`, allows suppression of the log message, simultaneous `colorSpace` setting and calls update methods.

setBorderRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set `obj._borderColor.rgb`

setColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Sets both the line and fill to be the same color.

setContrast (*newContrast*, *operation=""*, *log=None*)

Usually you can use ‘`stim.attribute = value`’ syntax instead, but use this method if you need to suppress the log message

setDKL (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth*, *operation=""*, *log=None*)

Usually you can use ‘`stim.attribute = value`’ syntax instead, but use this method if you need to suppress the log message

setEnd (*end*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setFillColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOpacity (*newOpacity*, *operation=""*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setPos (*newPos*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize (*newSize*, *operation=""*, *units=None*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setStart (*start*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setVertices (*value=None*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

start

tuple, list or 2x1 array.

Specifies the position of the start of the line. *Operations* supported.

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property vertices

A list of lists or a numpy array (Nx2) specifying xy positions of each vertex, relative to the center of the field.

Assigning to vertices can be slow if there are many vertices.

Operations supported with *.setVertices()*.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.15 MovieStim

Attributes

<code>MovieStim(win[, filename, movieLib, units, ...])</code>	Class for presenting movie clips as stimuli.
<code>MovieStim.win</code>	The <i>Window</i> object in which the stimulus will be rendered by default.
<code>MovieStim.units</code>	
<code>MovieStim.pos</code>	The position of the center of the stimulus in the stimulus <i>units</i>
<code>MovieStim.ori</code>	The orientation of the stimulus (in degrees).
<code>MovieStim.size</code>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<code>MovieStim.opacity</code>	Determines how visible the stimulus is relative to background.
<code>MovieStim.name</code>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<code>MovieStim.autoLog</code>	Whether every change in this stimulus should be auto logged.
<code>MovieStim.draw([win])</code>	Draw the current frame to a particular window.
<code>MovieStim.autoDraw</code>	Determines whether the stimulus should be automatically drawn on every frame flip.
<code>MovieStim.loadMovie(filename)</code>	Load a movie file from disk.
<code>MovieStim.play([log])</code>	Start or continue a paused movie from current position.
<code>MovieStim.seek(timestamp[, log])</code>	Seek to a particular timestamp in the movie.
<code>MovieStim.pause([log])</code>	Pause the current point in the movie.
<code>MovieStim.stop([log])</code>	Stop the current point in the movie (sound will stop, current frame will not advance).

Details

```
class psychopy.visual.MovieStim(win, filename="", movieLib='ffpyplayer', units='pix',
    size=None, pos=0.0, 0.0, ori=0.0, anchor='center',
    flipVert=False, flipHoriz=False, color=1.0, 1.0, 1.0, colorSpace='rgb', opacity=1.0, contrast=1, volume=1.0, name="",
    loop=False, autoLog=True, depth=0.0, noAudio=False,
    interpolate=True, autoStart=True)
```

Class for presenting movie clips as stimuli.

Parameters

- **win** (*Window*) – Window the video is being drawn to.
- **filename** (*str*) – Name of the file or stream URL to play. If an empty string, no file will be loaded on initialization but can be set later.
- **movieLib** (*str* or *None*) – Library to use for video decoding. By default, the ‘preferred’ library by PsychoPy developers is used. Default is ‘ffpyplayer’. An alert is raised if you are not using the preferred player.
- **units** (*str*) – Units to use when sizing the video frame on the window, affects how *size* is interpreted.
- **size** (*ArrayLike* or *None*) – Size of the video frame on the window in *units*. If *None*, the native size of the video will be used.

- **flipVert** (*bool*) – If *True* then the movie will be top-bottom flipped.
- **flipHoriz** (*bool*) – If *True* then the movie will be right-left flipped.
- **volume** (*int or float*) – If specifying an *int* the nominal level is 100, and 0 is silence. If a *float*, values between 0 and 1 may be used.
- **loop** (*bool*) – Whether to start the movie over from the beginning if draw is called and the movie is done. Default is *False*.
- **autoStart** (*bool*) – Automatically begin playback of the video when *flip()* is called.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`

`_drawRectangle()`

Draw the video frame to the window.

This is called by the *draw()* method to blit the video to the display window.

`_freeBuffers()`

Free texture and pixel buffers. Call this when tearing down this class or if a movie is stopped.

`_getDesiredRGB(rgb, colorSpace, contrast)`

Convert color to RGB while adding contrast. Requires `self.rgb`, `self.colorSpace` and `self.contrast`

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_pixelTransfer()`

Copy pixel data from video frame to texture.

`_selectWindow(win)`

Switch drawing to the specified window. Calls the window's `_setCurrent()` method which handles the switch.

`_set(attrib, val, op="", log=None)`

DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

`_setupTextureBuffers()`

Setup texture buffers which hold frame data. This creates a 2D RGB texture and pixel buffer. The pixel buffer serves as the store for texture color data. Each frame, the pixel buffer memory is mapped and frame data is copied over to the GPU from the decoder.

This is called every time a video file is loaded. The `_freeBuffers` method is called in this routine prior to creating new buffers, so it's safe to call this right after loading a new movie without having to `_freeBuffers` first.

`_updateList()`

The user shouldn't need this method since it gets called after every call to `.set()` Chooses between using and not using shaders each call.

`_updateVertices()`

Sets `Stim.verticesPix` and `._borderPix` from `pos`, `size`, `ori`, `flipVert`, `flipHoriz`

property anchor

autoDraw

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

autoLog

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property autoStart

Start playback when *.draw()* is called (*bool*).

property backColor

Alternative way of setting *fillColor*

property backColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property borderColor

property borderColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or *None*

For strings and hex values this is not needed. If *None* the default *colorSpace* for the stimulus is used (defined during initialisation).

Please note that changing *colorSpace* does not change stimulus parameters. Thus you usually want to specify *colorSpace* before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains (*x, y=None, units=None*)

Returns True if a point *x,y* is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such as a *Mouse*.**

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (*pos*), and orientation (*ori*) settings (and by the vertices for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

property **contrast**

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw (*win=None*)

Draw the current frame to a particular window.

The current position in the movie will be determined automatically. This method should be called on every frame that the movie is meant to appear. If *.autoStart==True* the video will begin playing when this is called.

Parameters *win* (*Window* or *None*) – Window the video is being drawn to. If *None*, the window specified at initialization will be used instead.

Returns *True* if the frame was updated this draw call.

Return type *bool*

property **duration**

Duration of the loaded video in seconds (*float*). Not valid unless the video has been started.

fastForward (*seconds=5, log=True*)

Fast-forward the video.

Parameters

- **seconds** (*float*) – Time in seconds to fast forward from the current position. Default is 5 seconds.
- **log** (*bool*) – Log this event.

Returns Timestamp at new position after fast forwarding the video.

Return type `float`

property filename

File name for the loaded video (*str*).

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current `colorSpace`. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
```

(continues on next page)

(continued from previous page)

```
stim.color = 255 # interpreted as (255, 255, 255)
                # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
                 # space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

property fps

Movie frames per second (*float*).

property frameIndex

Current frame index being displayed (*int*).

property frameRate

Frame rate of the movie in Hertz (*float*).

property frameSize

Size of the video (*w, h*) in pixels (*tuple*). Alias of *videoSize*.

property frameTexture

Texture ID for the current video frame (*GLuint*). You can use this as a video texture. However, you must periodically call *updateVideoFrame* to keep this up to date.

getCurrentFrameNumber ()

Get the current movie frame number (*int*), same as *frameIndex*.

getFPS ()

Movie frames per second.

Returns Nominal number of frames to be displayed per second.

Return type *float*

getPercentageComplete ()

Provides a value between 0.0 and 100.0, indicating the amount of the movie that has been already played (*float*).

property height**property isFinished**

True if the video is finished (*bool*).

property isNotStarted

True if the video may not have started yet (*bool*). This status is given after a video is loaded and play has yet to be called.

property isPaused

True if the video is presently paused (*bool*).

property isPlaying

True if the video is presently playing (*bool*).

property isStopped

True if the video is stopped (*bool*).

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

loadMovie (*filename*)

Load a movie file from disk.

Parameters filename (*str*) – Path to movie file. Must be a format that FFmpeg supports.

property loopCount

Number of loops completed since playback started (*int*). Incremented each time the movie begins another loop.

Examples

Compute how long a looping video has been playing until now:

```
totalMovieTime = (mov.loopCount + 1) * mov.pts
```

property muted

True if the stream audio is muted (*bool*).

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. `visual.TextStim(win)` will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, `shapeContains.py`

pause (*log=True*)

Pause the current point in the movie. The image of the last frame will persist on-screen until *play()* or *stop()* are called.

Parameters `log (bool)` – Log this event.

play (*log=True*)

Start or continue a paused movie from current position.

Parameters `log (bool)` – Log the play event.

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

property pts

Presentation timestamp of the most recent frame (*float*).

This value corresponds to the time in movie/stream time the frame is scheduled to be presented.

replay (*autoStart=True, log=True*)

Replay the movie from the beginning.

Parameters

- **autoStart** (*bool*) – Start playback immediately. If *False*, you must call *play()* afterwards to initiate playback.
- **log** (*bool*) – Log this event.

Notes

- This tears down the current media player instance and creates a new one. Similar to calling *stop()* and *loadMovie()*. Use *seek(0.0)* if you would like to restart the movie without reloading.

rewind (*seconds=5, log=True*)

Rewind the video.

Parameters

- **seconds** (*float*) – Time in seconds to rewind from the current position. Default is 5 seconds.
- **log** (*bool*) – Log this event.

Returns Timestamp after rewinding the video.

Return type *float*

seek (*timestamp, log=True*)

Seek to a particular timestamp in the movie.

Parameters

- **timestamp** (*float*) – Time in seconds.
- **log** (*bool*) – Log this event.

setAnchor (*value, log=None*)

setAutoDraw (*value, log=None*)

Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True, log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setBackColor (*color, colorSpace=None, operation="", log=None*)

setBackRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setBorderRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor (*color, colorSpace=None, operation="", log=None*)

setContrast (*newContrast, operation="", log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setDKL (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth, operation="", log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setFillColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setLineRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setMovie (*value*)

setOpacity (*newOpacity*, *operation=""*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos (*newPos*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize (*newSize*, *operation=""*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

stop (*log=True*)

Stop the current point in the movie (sound will stop, current frame will not advance). Once stopped the movie cannot be restarted - it must be loaded again.

Use *pause()* instead if you may need to restart the movie.

Parameters `log` (*bool*) – Log this event.

property units

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

updateVideoFrame ()

Update the present video frame. The next call to *draw*() will make the retrieved frame appear.

Returns If *True*, the video texture has been updated and the frame index is advanced by one. If *False*, the last frame should be kept on-screen.

Return type `bool`

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property videoSize

Size of the video (*w*, *h*) in pixels (*tuple*). Returns (*0*, *0*) if no video is loaded.

property volume

Volume for the audio track for this movie (*int* or *float*).

volumeDown (*amount=0.05*)

Decrease the volume by a fixed amount.

Parameters `amount` (*float* or *int*) – Amount to decrease the volume relative to the current volume.

volumeUp (*amount=0.05*)

Increase the volume by a fixed amount.

Parameters `amount` (*float* or *int*) – Amount to increase the volume relative to the current volume.

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (*win1*, *win2* and *stim*):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.16 NoiseStim

Attributes

Details

class psychopy.visual.NoiseStim(*args, **kwargs)

A stimulus with 2 textures: a random noise sample and a mask

Example:

```
noise1 = noise = visual.NoiseStim(
    win=win, name='noise', units='pix',
    noiseImage='testImg.jpg', mask='circle',
    ori=1.0, pos=(0, 0), size=(512, 512), sf=None, phase=0,
    color=[1,1,1], colorSpace='rgb', opacity=1, blendmode='add',
    ↪contrast=1.0,
    texRes=512, filter='None', imageComponent='Phase'
    noiseType='Gabor', noiseElementSize=4, noiseBaseSf=32.0/512,
    noiseBW=1.0, noiseBWO=30, noiseFractalPower=-1, noiseFilterLower=3/
    ↪512, noiseFilterUpper=8.0/512.0,
    noiseFilterOrder=3.0, noiseClip=3.0, filter=False,
    ↪interpolate=False, depth=-1.0)
# gives a circular patch of noise made up of scattered Gabor elements with peak
    ↪frequency = 32.0/512 cycles per pixel,
# orientation = 0 , frequency bandwidth = 1 octave and orientation bandwidth 30
    ↪degrees
```

Types of noise available

- Binary, Normal, Uniform - pixel based noise samples drawn from a binary (blank and white), normal or uniform distribution respectively. Binary noise is always exactly zero mean, Normal and Uniform are approximately so. Parameters:
 - noiseElementSize - (can be a tuple) defines the size of the noise elements in the components units.
 - noiseClip the values in normally distributed noise are divided by noiseClip to limit excessively high or low values. However, values can still go out of range -1 to 1 which will throw a soft error message high values of noiseClip are recommended if using 'Normal'
- **Gabor, Isotropic:** Effectively a dense scattering of Gabor elements with random amplitude and fixed orientation for Gabor or random orientation for Isotropic noise. In practice the desired amplitude spectrum for the noise is built in Fourier space with a random phase spectrum. DC term is set to zero - ie zero mean. Parameters:
 - noiseBaseSf - centre spatial frequency in the component units.
 - noiseBW - spatial frequency bandwidth full width half height in octaves.
 - ori - center orientation for Gabor noise (works as for gratingStim so twists the final image at render time).
 - noiseBWO - orientation bandwidth for Gabor noise full width half height in degrees.

- noiseOri - alternative center orientation for Gabor which sets the orientation during the image build rather than at render time. Useful for setting the orientation of a filter to be applied to some other noise type with a different base orientation.
- **Filtered** - A white noise sample that has been filtered with a low, high or bandpass Butterworth filter. The initial sample can have its spectrum skewed towards low or high frequencies. The contrast of the noise falls by half its maximum (3dB) at the cutoff frequencies. Parameters:
 - noiseFilterUpper - upper cutoff frequency - if greater than texRes/2 cycles per image low pass filter used.
 - noiseFilterLower - Lower cutoff frequency - if zero low pass filter used.
 - noiseFilterOrder - The order of the filter controls the steepness of the falloff outside the passband is zero no filter is applied.
 - noiseFractalPower - $\text{spectrum} = f^{\text{noiseFractalPower}}$ - determines the spatial frequency bias of the initial noise sample. 0 = flat spectrum, negative = low frequency bias, positive = high frequency bias, -1 = fractal or brownian noise.
 - noiseClip - determines clipping values and rescaling factor such that final rms contrast is close to that requested by contrast parameter while keeping pixel values in range -1, 1.
- **White** - A short cut to obtain noise with a flat, unfiltered spectrum. In practice the desired amplitude spectrum is built in the Fourier Domain with a random phase spectrum. DC term is set to zero - ie zero mean Note despite name the noise contains all grey levels. Parameters:
 - noiseClip - determines clipping values and rescaling factor such that final rms contrast is close to that requested by contrast parameter while keeping pixel values in range -1, 1.
- **Image**: A noise sample whose spatial frequency spectrum is taken from the supplied image. In practice the desired amplitude spectrum is taken from the image and paired with a random phase spectrum. DC term is set to zero - ie zero mean. Parameters:
 - noiseImage name of ndarray or image file from which to take spectrum - should be same size as largest side requested for component if units is pix or texRes x texRes otherwise
 - imageComponent: 'Phase' randomizes the phase spectrum leaving the amplitude spectrum untouched. 'Amplitude' randomizes the amplitude spectrum leaving the phase spectrum untouched - retains spatial structure of image. 'Neither' keeps the image as is - but you can now apply a spatial filter to the image.
 - noiseClip - determines clipping values and rescaling factor such that final rms contrast is close to that requested by contrast parameter while keeping pixel values in range -1, 1.

Filter parameter

- Butterworth: a spectral filter defined by the filtered noise parameters will be applied to the other noise types.
- Gabor: a spectral filter defined by the Gabor noise parameters will be applied to the other noise types.
- Isotropic: then a spectral filter defined by the Isotropic noise parameters will be applied to the other noise types.

Updating noise samples and timing

The noise is rebuilt at next call of the draw function whenever a parameter starting 'noise' is notionally changed even if the value does not actually change every time. eg. setting a parameter to update every frame will cause a new noise sample on every frame but see below. A rebuild can also be forced at any time using the buildNoise() function. The updateNoise() function can be used at any time to produce a new random sample of noise without doing a full build. ie it is quicker than a full build. Both buildNoise and updateNoise can be slow for large samples. Samples of Binary, Normal or Uniform noise can usually be made at frame rate using noiseUpdate.

Updating or building other noise types at frame rate may result in dropped frames. An alternative is to build a large sample of noise at the start of the routine and place it off the screen then cut a samples out of this at random locations and feed that as a numpy array into the texture of a visible gratingStim.

Notes on size If units = pix and noiseType = Binary, Normal or Uniform will make noise sample of requested size. If units = pix and noiseType is Gabor, Isotropic, Filtered, White, Coloured or Image will make square noise sample with side length equal that of the largest dimetions requested. if units is not pix will make square noise sample with side length equal to texRes then rescale to present.

Notes on interpolation For pixel based noise interpolation = nearest is usually best. For other noise types linear is better if the size of the noise sample does not match the final size of the image well.

Notes on frequency Frequencies for cutoffs etc are converted between units for you but can be counter intuitive. 1/size is always 1 cycle per image. For the sf (final spatial frequency) parameter itself 1/size (or None for units pix) will faithfully represent the image without further scaling.

Filter cutoff and Gabor/Isotropic base frequencies should not be too high you should aim to keep them below 0.5 c/pixel on the screen. The function will produce an error when it can't draw the stimulus in the buffer but it may still be wrong when displayed.

Notes on orientation and phase The ori parameter twists the final image so the samples in noiseType Binary, Normal or Uniform will no longer be aligned to the sides of the monitor if ori is not a multiple of 90. Most other noise types look broadly the same for all values of ori but the specific sample shown can be made to rotate by changing ori. The dominant orientation for Gabor noise is determined by ori at render time, not before.

The phase parameter similarly shifts the sample around within the display window at render time and will not choose new random phases for the noise sample.

9.3.17 ObjMeshStim

Attributes

<code>ObjMeshStim(win, objFile[, pos, ori, ...])</code>	Class for loading and presenting 3D stimuli in the Wavefront OBJ format.
---	--

Details

```
class psychopy.visual.ObjMeshStim(win, objFile, pos=0, 0, 0, ori=0, 0, 0, 1, useMaterial=None,
                                  loadMtlLib=True, color=0.0, 0.0, 0.0, colorSpace='rgb',
                                  contrast=1.0, opacity=1.0, name="", autoLog=True)
```

Class for loading and presenting 3D stimuli in the Wavefront OBJ format.

Calling the *draw* method will render the mesh to the current buffer. The render target (FBO or back buffer) must have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting *useLights=True* before drawing the stimulus).

Vertex positions, texture coordinates, and normals are loaded and packed into a single vertex buffer object (VBO). Vertex array objects (VAO) are created for each material with an index buffer referencing vertices assigned that material in the VBO. For maximum performance, keep the number of materials per object as low as possible, as switching between VAOs has some overhead.

Material attributes are read from the material library file (*.MTL) associated with the *.OBJ file. This file will be automatically searched for and read during loading. Afterwards you can edit material properties by accessing the data structure of the *materials* attribute.

Keep in mind that OBJ shapes are rigid bodies, the mesh itself cannot be deformed during runtime. However,

meshes can be positioned and rotated as desired by manipulating the *RigidBodyPose* instance accessed through the *thePose* attribute.

Warning: Loading an *.OBJ file is a slow process, be sure to do this outside of any time-critical routines! This class is experimental and may result in undefined behavior.

Examples

Loading an *.OBJ file from a disk location:

```
myObjStim = ObjMeshStim(win, '/path/to/file/model.obj')
```

Parameters

- **win** (*~psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- **size** (*tuple or float*) – Dimensions of the mesh. If a single value is specified, the plane will be a square. Provide a tuple of floats to specify the width and length of the box (eg. *size=(0.2, 1.3)*).
- **pos** (*array_like*) – Position vector $[x, y, z]$ for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion $[x, y, z, w]$ where x, y, z are imaginary and w is real. If you prefer specifying rotations in axis-angle format, call *setOriAxisAngle* after initialization. By default, the plane is oriented with normal facing the +Z axis of the scene.
- **useMaterial** (*PhongMaterial, optional*) – Material to use for all sub-meshes. The material can be configured by accessing the *material* attribute after initialization. If no material is specified, *color* will modulate the diffuse and ambient colors for all meshes in the model. If *loadMtlLib* is *True*, this value should be *None*.
- **loadMtlLib** (*bool*) – Load materials from the MTL file associated with the mesh. This will override *useMaterial* if it is *None*. The value of *materials* after initialization will be a dictionary where keys are material names and values are materials. Any textures associated with the model will be loaded as per the material requirements.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

_createVAO (*vertices, textureCoords, normals, faces*)

Create a vertex array object for handling vertex attribute data.

_getDesiredRGB (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires *self.rgb*, *self.colorSpace* and *self.contrast*

_loadMtlLib (*mtlFile*)

Load a material library associated with the OBJ file. This is usually called by the constructor for this class.

Parameters **mtlFile** (*str*) – Path to MTL file.

_selectWindow (*win*)

Switch drawing to the specified window. Calls the window's *_setCurrent()* method which handles the switch.

`_updateList()`

The user shouldn't need this method since it gets called after every call to `.set()`. Chooses between using and not using shaders each call.

property `anchor`

property `backColor`

Alternative way of setting `fillColor`

property `backColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object.

property `backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property `borderColor`

property `borderColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object

property `borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use `obj._borderColor.rgb`

Type DEPRECATED

property `color`

Alternative way of setting `foreColor`.

property `colorSpace`

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default `colorSpace` for the stimulus is used (defined during initialisation).

Please note that changing `colorSpace` does not change stimulus parameters. Thus you usually want to specify `colorSpace` before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property `contrast`

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

draw (*win=None*)

Draw the mesh.

Parameters *win* (~*psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current *colorSpace*. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property **foreColorSpace**

Deprecated, please use *colorSpace* to set color space for the entire object.

property **foreRGB**

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

getOri ()

getOriAxisAngle (*degrees=True*)

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the *ori* quaternion to and axis-angle representation.

Parameters **degrees** (*bool, optional*) – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

Returns Axis [*rx, ry, rz*] and angle.

Return type tuple

getPos ()

getRayIntersectBounds (*rayOrig, rayDir*)

Get the point which a ray intersects the bounding box of this mesh.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [*x, y, z*].
- **rayDir** (*array_like*) – Direction vector of the ray [*x, y, z*], should be normalized.

Returns Coordinate in world space of the intersection and distance in scene units from *rayOrig*.
Returns *None* if there is no intersection.

Return type `tuple`

property height

isVisible()

Check if the object is visible to the observer.

Test if a pose's bounding box or position falls outside of an eye's view frustum.

Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e. triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

Returns *True* if the object's bounding box is visible.

Return type `bool`

Examples

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```
if myStim.isVisible():
    myStim.draw()
```

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

property ori

Orientation quaternion (X, Y, Z, W).

property pos

Position vector (X, Y, Z).

setAnchor (*value*, *log=None*)

setBackColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setBackRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setBorderRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setContrast (*newContrast*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setFillColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor (*color, colorSpace=None, operation="", log=None*)

setLineRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOri (*ori*)

setOriAxisAngle (*axis, angle, degrees=True*)

Set the orientation of the 3D stimulus using an *axis* and *angle*. This sets the quaternion at *ori*.

Parameters

- **axis** (*array_like*) – Axis of rotation [rx, ry, rz].
- **angle** (*float*) – Angle of rotation.
- **degrees** (*bool, optional*) – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

setPos (*pos*)

setRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

property size

property thePose

The pose of the rigid body. This is a class which has *pos* and *ori* attributes.

units

None, 'norm', 'cm', 'deg', 'degFlat', 'degFlatPos', or 'pix'

If None then the current units of the *Window* will be used. See *Units for the window and stimuli* for explanation of other options.

Note that when you change units, you don't change the stimulus parameters and it is likely to change appearance. Example:

```
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5)

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

property vertices

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.18 PatchStim (deprecated)

class psychopy.visual.PatchStim(*args, **kwargs)

Deprecated (as of version 1.74.00): please use the *GratingStim* or the *ImageStim* classes.

The *GratingStim* has identical abilities to the *PatchStim* (but possibly different initial values) whereas the *ImageStim* is designed to be use for non-cyclic images (photographs, not gratings).

9.3.19 BlinnPhongMaterial

Attributes

<i>BlinnPhongMaterial</i> ([win, diffuseColor, ...])	Class representing a material using the Blinn-Phong lighting model.
--	---

Details

class psychopy.visual.BlinnPhongMaterial (win=None, diffuseColor=0.5, 0.5, 0.5, specularColor=- 1.0, - 1.0, - 1.0, ambientColor=- 1.0, - 1.0, - 1.0, emissionColor=- 1.0, - 1.0, - 1.0, shininess=10.0, colorSpace='rgb', diffuseTexture=None, opacity=1.0, contrast=1.0, face='front')

Class representing a material using the Blinn-Phong lighting model.

This class stores material information to modify the appearance of drawn primitives with respect to lighting, such as color (diffuse, specular, ambient, and emission), shininess, and textures. Simple materials are intended to work with features supported by the fixed-function OpenGL pipeline.

If shaders are enabled, the colors of objects will appear different than without. This is due to the lighting/material colors being computed on a per-pixel basis, and the formulation of the lighting model. The Phong shader determines the ambient color/intensity by adding up both the scene and light ambient colors, then multiplies them by the diffuse color of the material, as the ambient light's color should be a product of the surface reflectance (albedo) and the light color (the ambient light needs to reflect off something to be visible). Diffuse reflectance is Lambertian, where the cosine angle between the incident light ray and surface normal determines color. The size of specular highlights are related to the *shininess* factor which ranges from 1.0 to 128.0. The greater this number, the tighter the specular highlight making the surface appear smoother. If shaders are not being used, specular highlights will be computed using the Phong lighting model. The emission color is optional, it simply adds to the color of every pixel much like ambient lighting does. Usually, you would not really want this, but it can be used to add bias to the overall color of the shape.

If there are no lights in the scene, the diffuse color is simply multiplied by the scene and material ambient color to give the final color.

Lights are attenuated (fall-off with distance) using the formula:

```
attenuationFactor = 1.0 / (k0 + k1 * distance + k2 * pow(distance, 2))
```

The coefficients for attenuation can be specified by setting *attenuation* in the lighting object. Values $k0=1.0$, $k1=0.0$, and $k2=0.0$ results in a light that does not fall-off with distance.

Warning: This class is experimental and may result in undefined behavior.

Parameters

- **win** (*~psychopy.visual.Window* or *None*) – Window this material is associated with, required for shaders and some color space conversions.
- **diffuseColor** (*array_like*) – Diffuse material color (r, g, b, a) with values between 0.0 and 1.0.
- **specularColor** (*array_like*) – Specular material color (r, g, b, a) with values between 0.0 and 1.0.
- **ambientColor** (*array_like*) – Ambient material color (r, g, b, a) with values between 0.0 and 1.0.
- **emissionColor** (*array_like*) – Emission material color (r, g, b, a) with values between 0.0 and 1.0.
- **shininess** (*float*) – Material shininess, usually ranges from 0.0 to 128.0.
- **colorSpace** (*float*) – Color space for *diffuseColor*, *specularColor*, *ambientColor*, and *emissionColor*.
- **diffuseTexture** (*TexImage2D*) –
- **opacity** (*float*) – Opacity of the material. Ranges from 0.0 to 1.0 where 1.0 is fully opaque.
- **contrast** (*float*) – Contrast of the material colors.
- **face** (*str*) – Face to apply material to. Values are *front*, *back* or *both*.

- **textures** (*dict, optional*) – Texture maps associated with this material. Textures are specified as a list. The index of textures in the list will be used to set the corresponding texture unit they are bound to.

property ambientColor

Ambient color of the material.

property ambientRGB

Diffuse color of the material.

begin (*useTextures=True*)

Use this material for successive rendering calls.

Parameters useTextures (*bool*) – Enable textures.

property diffuseColor

Diffuse color of the material.

property diffuseRGB

Diffuse color of the material.

property diffuseTexture

Diffuse color of the material.

property emissionColor

Emission color of the material.

property emissionRGB

Diffuse color of the material.

end (*clear=True*)

Stop using this material.

Must be called after *begin* before using another material or else later drawing operations may have undefined behavior.

Upon returning, *GL_COLOR_MATERIAL* is enabled so material colors will track the current *glColor*.

Parameters clear (*bool*) – Overwrite material state settings with default values. This ensures material colors are set to OpenGL defaults. You can forgo clearing if successive materials are used which overwrite *glMaterialfv* values for *GL_DIFFUSE*, *GL_SPECULAR*, *GL_AMBIENT*, *GL_EMISSION*, and *GL_SHININESS*. This reduces a bit of overhead if there is no need to return to default values intermittently between successive material *begin* and *end* calls. Textures and shaders previously enabled will still be disabled.

property shininess

property specularColor

Specular color of the material.

property specularRGB

Diffuse color of the material.

9.3.20 psychopy.visual.Pie

Stimulus class for drawing semi-circles and wedges.

Overview

<code>Pie(win[, radius, start, end, edges, units, ...])</code>	Creates a pie shape which is a circle with a wedge cut-out.
<code>Pie.start</code>	Start angle of the slice/wedge in degrees (<i>float</i> or <i>int</i>).
<code>Pie.end</code>	End angle of the slice/wedge in degrees (<i>float</i> or <i>int</i>).
<code>Pie.radius</code>	Radius of the shape in <i>units</i> (<i>float</i> or <i>int</i>).
<code>Pie.units</code>	
<code>Pie.lineWidth</code>	Width of the line in pixels .
<code>Pie.lineColor</code>	Alternative way of setting <i>borderColor</i> .
<code>Pie.lineColorSpace</code>	Deprecated, please use <i>colorSpace</i> to set color space for the entire object
<code>Pie.fillColor</code>	Set the fill color for the shape.
<code>Pie.fillColorSpace</code>	Deprecated, please use <i>colorSpace</i> to set color space for the entire object.
<code>Pie.pos</code>	The position of the center of the stimulus in the stimulus <i>units</i>
<code>Pie.size</code>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<code>Pie.ori</code>	The orientation of the stimulus (in degrees).
<code>Pie.opacity</code>	Determines how visible the stimulus is relative to background.
<code>Pie.contrast</code>	A value that is simply multiplied by the color.
<code>Pie.depth</code>	DEPRECATED, depth is now controlled simply by drawing order.
<code>Pie.interpolate</code>	If <i>True</i> the edge of the line will be anti-aliased.
<code>Pie.lineRGB</code>	Legacy property for setting the border color of a stimulus in RGB, instead use <i>obj._borderColor.rgb</i>
<code>Pie.fillRGB</code>	Legacy property for setting the fill color of a stimulus in RGB, instead use <i>obj._fillColor.rgb</i>
<code>Pie.name</code>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<code>Pie.autoLog</code>	Whether every change in this stimulus should be auto logged.
<code>Pie.autoDraw</code>	Determines whether the stimulus should be automatically drawn on every frame flip.
<code>Pie.color</code>	Set the color of the shape.
<code>Pie.colorSpace</code>	The name of the color space currently being used

Details

```
class psychopy.visual.pie.Pie(win, radius=0.5, start=0.0, end=90.0, edges=32, units="",
    lineWidth=1.5, lineColor=None, lineColorSpace='rgb', fillColor=None,
    fillColorSpace='rgb', pos=0, 0, size=1.0, ori=0.0, opacity=1.0,
    contrast=1.0, depth=0, interpolate=True, lineRGB=False, fillRGB=False,
    name=None, autoLog=None, autoDraw=False, color=None, colorSpace=None)
```

Creates a pie shape which is a circle with a wedge cut-out.

This shape is sometimes referred to as a Pac-Man shape which is often used for creating Kanizsa figures. However, the shape can be adapted for other uses.

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **radius** (*float or int*) – Radius of the shape. Avoid using *size* for adjusting figure dimensions if radius != 0.5 which will result in undefined behavior.
- **start** (*float or int*) – Start and end angles of the filled region of the shape in degrees. Shapes are filled counter clockwise between the specified angles.
- **end** (*float or int*) – Start and end angles of the filled region of the shape in degrees. Shapes are filled counter clockwise between the specified angles.
- **edges** (*int*) – Number of edges to use when drawing the figure. A greater number of edges will result in smoother curves, but will require more time to compute.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **lineWidth** (*float*) – Width of the shape's outline.
- **lineColor** (*array_like, str, Color or None*) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **fillColor** (*array_like, str, Color or None*) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **lineColorSpace** (*str*) – Colorspace to use for the outline and fill. These change how the values passed to *lineColor* and *fillColor* are interpreted. *Deprecated*. Please use *colorSpace* to set both outline and fill colorspace. These arguments may be removed in a future version.
- **fillColorSpace** (*str*) – Colorspace to use for the outline and fill. These change how the values passed to *lineColor* and *fillColor* are interpreted. *Deprecated*. Please use *colorSpace* to set both outline and fill colorspace. These arguments may be removed in a future version.
- **pos** (*array_like*) – Initial position (*x, y*) of the shape on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is (0.0, 0.0) which results in no translation.
- **size** (*array_like, float, int or None*) – Width and height of the shape as (*w, h*) or [*w, h*]. If a single value is provided, the width and height will be set to the same specified value. If *None* is specified, the *size* will be set with values passed to *width* and *height*.

- **ori** (*float*) – Initial orientation of the shape in degrees about its origin. Positive values will rotate the shape clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.
- **contrast** (*float*) – Contrast level of the shape (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the shape when *autoDraw* is enabled. *DEPRECATED*
- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing shape outlines. This produces a smoother (less-pixelated) outline of the shape.
- **lineRGB** (array_like, *Color* or None) – *Deprecated*. Please use *lineColor* and *fillColor*. These arguments may be removed in a future version.
- **fillRGB** (array_like, *Color* or None) – *Deprecated*. Please use *lineColor* and *fillColor*. These arguments may be removed in a future version.
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the *draw()* method.
- **color** (array_like, str, *Color* or None) – Sets both the initial *lineColor* and *fillColor* of the shape.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.

start, end

Start and end angles of the filled region of the shape in degrees. Shapes are filled counter clockwise between the specified angles.

Type float or int

radius

Radius of the shape. Avoid using *size* for adjusting figure dimensions if radius != 0.5 which will result in undefined behavior.

Type float or int

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

static _calcEquilateralVertices (*edges, radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

_calcPosRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*.

_calcSizeRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*

`_calcVertices ()`
 Calculate the required vertices for the figure.

`_getDesiredRGB (rgb, colorSpace, contrast)`
 Convert color to RGB while adding contrast. Requires `self.rgb`, `self.colorSpace` and `self.contrast`

`_getPolyAsRendered ()`
 DEPRECATED. Return a list of vertices as rendered.

`_selectWindow (win)`
 Switch drawing to the specified window. Calls the window's `_setCurrent()` method which handles the switch.

`_set (attrib, val, op="", log=None)`
 DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

`_updateList ()`
 The user shouldn't need this method since it gets called after every call to `.set()` Chooses between using and not using shaders each call.

`_updateVertices ()`
 Sets `Stim.verticesPix` and `._borderPix` from `pos`, `size`, `ori`, `flipVert`, `flipHoriz`

`autoDraw`
 Determines whether the stimulus should be automatically drawn on every frame flip.
 Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`
 Whether every change in this stimulus should be auto logged.
 Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`property backColor`
 Alternative way of setting `fillColor`

`property backColorSpace`
 Deprecated, please use `colorSpace` to set color space for the entire object.

`property backRGB`
 Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`
Type DEPRECATED

`property borderColorSpace`
 Deprecated, please use `colorSpace` to set color space for the entire object

`property borderRGB`
 Legacy property for setting the border color of a stimulus in RGB, instead use `obj._borderColor.rgb`
Type DEPRECATED

`closeShape`
 Should the last vertex be automatically connected to the first?
 If you're using *Polygon*, *Circle* or *Rect*, `closeShape=True` is assumed and shouldn't be changed.

`color`
 Set the color of the shape. Sets both `fillColor` and `lineColor` simultaneously if applicable.

`property colorSpace`
 The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains (*x, y=None, units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such** as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (*pos*), and orientation (*ori*) settings (and by the vertices for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw (*win=None, keepMatrix=False*)

Draw the stimulus in its relevant window.

You must call this method after every `MyWin.flip()` if you want the stimulus to appear on that frame and then update the screen again.

end

End angle of the slice/wedge in degrees (*float* or *int*).

Operations supported.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as `True` to flip or `False` to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current `colorSpace`. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
                 # space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

interpolate

If *True* the edge of the line will be anti-aliased.

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

lineWidth

Width of the line in **pixels**.

Operations supported.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, `shapeContains.py`

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

radius

Radius of the shape in *units* (*float* or *int*).

Operations supported.

setAutoDraw (*value*, *log=None*)

Sets `autoDraw`. Usually you can use ‘`stim.attribute = value`’ syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True*, *log=None*)

Usually you can use ‘`stim.attribute = value`’ syntax instead, but use this method if you need to suppress the log message.

setBackRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set `obj._fillColor.rgb`

setBorderColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for `fillColor`, allows suppression of the log message, simultaneous `colorSpace` setting and calls update methods.

setBorderRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set `obj._borderColor.rgb`

setColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Sets both the line and fill to be the same color.

setContrast (*newContrast*, *operation=""*, *log=None*)

Usually you can use ‘`stim.attribute = value`’ syntax instead, but use this method if you need to suppress the log message

setDKL (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setEnd (*end*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setFillColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for fillColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setFillRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOpacity (*newOpacity*, *operation=""*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos (*newPos*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setRadius (*end*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setSize (*newSize*, *operation=""*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setStart (*start*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setVertices (*value=None*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

start

Start angle of the slice/wedge in degrees (*float* or *int*).

Operations supported.

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.21 PlaneStim

Attributes

<code>PlaneStim(win[, size, pos, ori, color, ...])</code>	Class for drawing planes.
---	---------------------------

Details

```
class psychopy.visual.PlaneStim(win, size=0.5, 0.5, pos=0.0, 0.0, 0.0, ori=0.0, 0.0, 0.0, 1.0,
                                color=0.0, 0.0, 0.0, colorSpace='rgb', contrast=1.0, opacity=1.0, useMaterial=None, textureScale=None, name="", autoLog=True)
```

Class for drawing planes.

Draws a plane with dimensions specified by *size* (length, width) in scene units.

Calling the *draw* method will render the plane to the current buffer. The render target (FBO or back buffer) must have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting *useLights=True* before drawing the stimulus).

Warning: This class is experimental and may result in undefined behavior.

Parameters

- **win** (*~psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- **size** (*tuple or float*) – Dimensions of the mesh. If a single value is specified, the plane will be a square. Provide a tuple of floats to specify the width and length of the plane (eg. *size=(0.2, 1.3)*).
- **pos** (*array_like*) – Position vector $[x, y, z]$ for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion $[x, y, z, w]$ where x, y, z are imaginary and w is real. If you prefer specifying rotations in axis-angle format, call *setOriAxisAngle* after initialization. By default, the plane is oriented with normal facing the +Z axis of the scene.
- **useMaterial** (*PhongMaterial, optional*) – Material to use. The material can be configured by accessing the *material* attribute after initialization. If not material is specified, the diffuse and ambient color of the shape will track the current color specified by *glColor*.
- **colorSpace** (*str*) – Colorspace of *color* to use.
- **contrast** (*float*) – Contrast of the stimulus, value modulates the *color*.
- **opacity** (*float*) – Opacity of the stimulus ranging from 0.0 to 1.0. Note that transparent objects look best when rendered from farthest to nearest.
- **textureScale** (*array_like or float, optional*) – Scaling factors for texture coordinates (sx, sy). By default, a factor of 1 will have the entire texture cover the surface of the mesh. If a single number is provided, the texture will be scaled uniformly.
- **name** (*str*) – Name of this object for logging purposes.
- **autoLog** (*bool*) – Enable automatic logging on attribute changes.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

__createVAO (*vertices, textureCoords, normals, faces*)

Create a vertex array object for handling vertex attribute data.

__getDesiredRGB (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires *self.rgb*, *self.colorSpace* and *self.contrast*

__selectWindow (*win*)

Switch drawing to the specified window. Calls the window's *_setCurrent()* method which handles the switch.

__updateList ()

The user shouldn't need this method since it gets called after every call to *.set()* Chooses between using and not using shaders each call.

property anchor

property backColor

Alternative way of setting *fillColor*

property backColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property borderColor

property borderColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default *colorSpace* for the stimulus is used (defined during initialisation).

Please note that changing *colorSpace* does not change stimulus parameters. Thus you usually want to specify *colorSpace* before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'
```

(continues on next page)

(continued from previous page)

```
# Make it light green again
stim.color = (128, 255, 128)
```

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

draw (*win=None*)

Draw the stimulus.

This should work for stimuli using a single VAO and material. More complex stimuli with multiple materials should override this method to correctly handle that case.

Parameters *win* (~*psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz**property flipVert****property foreColor**

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.

- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other *Color spaces*.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property `foreColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object.

property `foreRGB`

Legacy property for setting the foreground color of a stimulus in RGB, instead use `obj._foreColor.rgb`

Type DEPRECATED

`getOri()`

`getOriAxisAngle` (*degrees=True*)

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the *ori* quaternion to and axis-angle representation.

Parameters `degrees` (*bool, optional*) – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

Returns Axis [*rx, ry, rz*] and angle.

Return type tuple

`getPos ()`

`getRayIntersectBounds (rayOrig, rayDir)`

Get the point which a ray intersects the bounding box of this mesh.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.

Returns Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type `tuple`

property height

`isVisible ()`

Check if the object is visible to the observer.

Test if a pose's bounding box or position falls outside of an eye's view frustum.

Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e. triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

Returns *True* if the object's bounding box is visible.

Return type `bool`

Examples

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```
if myStim.isVisible():
    myStim.draw()
```

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

property ori

Orientation quaternion (X, Y, Z, W).

property pos

Position vector (X, Y, Z).

setAnchor (*value*, *log=None*)

setBackColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setBackRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setBorderRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor (*color, colorSpace=None, operation="", log=None*)

setContrast (*newContrast, operation="", log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setDKL (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setFillColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setFillRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *foreColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setForeRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor (*color, colorSpace=None, operation="", log=None*)

setLineRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOri (*ori*)

setOriAxisAngle (*axis, angle, degrees=True*)

Set the orientation of the 3D stimulus using an *axis* and *angle*. This sets the quaternion at *ori*.

Parameters

- **axis** (*array_like*) – Axis of rotation [rx, ry, rz].
- **angle** (*float*) – Angle of rotation.
- **degrees** (*bool, optional*) – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

setPos (*pos*)

setRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

property size

property thePose

The pose of the rigid body. This is a class which has *pos* and *ori* attributes.

units

None, ‘norm’, ‘cm’, ‘deg’, ‘degFlat’, ‘degFlatPos’, or ‘pix’

If None then the current units of the *Window* will be used. See *Units for the window and stimuli* for explanation of other options.

Note that when you change units, you don't change the stimulus parameters and it is likely to change appearance. Example:

```
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5)

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

property vertices

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.22 psychopy.visual.Polygon

Stimulus class for drawing regular polygons.

Overview

<i>Polygon</i> (win[, edges, radius, units, ...])	Creates a regular polygon (triangles, pentagons, ...).
<i>Polygon.radius</i>	float, int, tuple, list or 2x1 array Radius of the Polygon (distance from the center to the corners).
<i>Polygon.edges</i>	Number of edges of the polygon.
<i>Polygon.units</i>	
<i>Polygon.lineWidth</i>	Width of the line in pixels .
<i>Polygon.lineColor</i>	Alternative way of setting <i>borderColor</i> .

continues on next page

Table 9.16 – continued from previous page

<i>Polygon.lineColorSpace</i>	Deprecated, please use <code>colorSpace</code> to set color space for the entire object
<i>Polygon.fillColor</i>	Set the fill color for the shape.
<i>Polygon.fillColorSpace</i>	Deprecated, please use <code>colorSpace</code> to set color space for the entire object.
<i>Polygon.pos</i>	The position of the center of the stimulus in the stimulus <i>units</i>
<i>Polygon.size</i>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<i>Polygon.ori</i>	The orientation of the stimulus (in degrees).
<i>Polygon.opacity</i>	Determines how visible the stimulus is relative to background.
<i>Polygon.contrast</i>	A value that is simply multiplied by the color.
<i>Polygon.depth</i>	DEPRECATED, depth is now controlled simply by drawing order.
<i>Polygon.interpolate</i>	If <i>True</i> the edge of the line will be anti-aliased.
<i>Polygon.lineRGB</i>	Legacy property for setting the border color of a stimulus in RGB, instead use <i>obj._borderColor.rgb</i>
<i>Polygon.fillRGB</i>	Legacy property for setting the fill color of a stimulus in RGB, instead use <i>obj._fillColor.rgb</i>
<i>Polygon.name</i>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<i>Polygon.autoLog</i>	Whether every change in this stimulus should be auto logged.
<i>Polygon.autoDraw</i>	Determines whether the stimulus should be automatically drawn on every frame flip.
<i>Polygon.color</i>	Set the color of the shape.
<i>Polygon.colorSpace</i>	The name of the color space currently being used

Details

```
class psychopy.visual.polygon.Polygon(win, edges=3, radius=0.5, units="", lineWidth=1.5,
lineColor=None, lineColorSpace=None, fillColor='white', fillColorSpace=None, pos=0, 0,
size=1.0, anchor=None, ori=0.0, opacity=None, contrast=1.0, depth=0, interpolate=True, lin-
eRGB=False, fillRGB=False, name=None, au-
toLog=None, autoDraw=False, color=None, col-
orSpace='rgb')
```

Creates a regular polygon (triangles, pentagons, ...).

This class is a special case of a `ShapeStim` that accepts the same parameters except `closeShape` and `vertices`.

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **edges** (*int*) – Number of sides for the polygon (for instance, `edges=3` will result in a triangle).

- **radius** (*float*) – Initial radius of the polygon in *units*. This specifies how far out to place the corners (vertices) of the shape.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **lineWidth** (*float*) – Width of the polygon’s outline.
- **lineColor** (array_like, str, *Color* or *None*) – Color of the shape’s outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **fillColor** (array_like, str, *Color* or *None*) – Color of the shape’s outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **lineColorSpace** (*str*) – Colorspace to use for the outline and fill. These change how the values passed to *lineColor* and *fillColor* are interpreted. *Deprecated*. Please use *colorSpace* to set both outline and fill colorspace. These arguments may be removed in a future version.
- **fillColorSpace** (*str*) – Colorspace to use for the outline and fill. These change how the values passed to *lineColor* and *fillColor* are interpreted. *Deprecated*. Please use *colorSpace* to set both outline and fill colorspace. These arguments may be removed in a future version.
- **pos** (*array_like*) – Initial position (*x*, *y*) of the shape on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is (0.0, 0.0) which results in no translation.
- **size** (*float* or *array_like*) – Initial scale factor for adjusting the size of the shape. A single value (*float*) will apply uniform scaling, while an array (*sx*, *sy*) will result in anisotropic scaling in the horizontal (*sx*) and vertical (*sy*) direction. Providing negative values to *size* will cause the shape being mirrored. Scaling can be changed by setting the *size* property after initialization. The default value is 1.0 which results in no scaling.
- **ori** (*float*) – Initial orientation of the shape in degrees about its origin. Positive values will rotate the shape clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.
- **contrast** (*float*) – Contrast level of the shape (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the stimulus when *autoDraw* is enabled.
- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing shape outlines. This produces a smoother (less-pixelated) outline of the shape.
- **lineRGB** (array_like, *Color* or *None*) – *Deprecated*. Please use *lineColor* and *fillColor*. These arguments may be removed in a future version.
- **fillRGB** (array_like, *Color* or *None*) – *Deprecated*. Please use *lineColor* and *fillColor*. These arguments may be removed in a future version.
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.

- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the *draw()* method.
- **color** (array_like, str, *Color* or *None*) – Sets both the initial *lineColor* and *fillColor* of the shape.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

static _calcEquilateralVertices (*edges, radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

_calcPosRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*.

_calcSizeRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*

_getDesiredRGB (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires *self.rgb*, *self.colorSpace* and *self.contrast*

_getPolyAsRendered ()

DEPRECATED. Return a list of vertices as rendered.

_selectWindow (*win*)

Switch drawing to the specified window. Calls the window's *_setCurrent()* method which handles the switch.

_set (*attrib, val, op="", log=None*)

DEPRECATED since 1.80.04 + 1. Use *setAttribute()* and *val2array()* instead.

_updateList ()

The user shouldn't need this method since it gets called after every call to *.set()* Chooses between using and not using shaders each call.

_updateVertices ()

Sets *Stim.verticesPix* and *._borderPix* from *pos*, *size*, *ori*, *flipVert*, *flipHoriz*

autoDraw

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

autoLog

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property backColor

Alternative way of setting *fillColor*

property backColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property borderColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use `obj._borderColor.rgb`

Type DEPRECATED

closeShape

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, `closeShape=True` is assumed and shouldn't be changed.

color

Set the color of the shape. Sets both *fillColor* and *lineColor* simultaneously if applicable.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default `colorSpace` for the stimulus is used (defined during initialisation).

Please note that changing `colorSpace` does not change stimulus parameters. Thus you usually want to specify `colorSpace` before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains (*x*, *y=None*, *units=None*)

Returns True if a point *x,y* is inside the stimulus' border.

Can accept variety of input options:

- two separate args, *x* and *y*
- one arg (list, tuple or array) containing two vals (*x,y*)
- **an object with a `getPos()` method that returns *x,y*, such as a *Mouse*.**

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (*pos*), and orientation (*ori*) settings (and by the vertices for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw (win=None, keepMatrix=False)

Draw the stimulus in its relevant window.

You must call this method after every `MyWin.flip()` if you want the stimulus to appear on that frame and then update the screen again.

edges

Number of edges of the polygon. Floats are rounded to int.

Operations supported.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current `colorSpace`. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property **foreColorSpace**

Deprecated, please use *colorSpace* to set color space for the entire object.

property **foreRGB**

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

property **interpolate**

If *True* the edge of the line will be anti-aliased.

property **lineColor**

Alternative way of setting *borderColor*.

property **lineColorSpace**

Deprecated, please use *colorSpace* to set color space for the entire object

property **lineRGB**

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

property **lineWidth**

Width of the line in **pixels**.

Operations supported.

property **name**

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

radius

float, int, tuple, list or 2x1 array Radius of the Polygon (distance from the center to the corners). May be a -2tuple or list to stretch the polygon asymmetrically.

Operations supported.

Usually there’s a setAttribute(value, log=False) method for each attribute. Use this if you want to disable logging.

setAutoDraw (*value, log=None*)

Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBackRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setBorderRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor (*color, colorSpace=None, operation="", log=None*)

Sets both the line and fill to be the same color.

setContrast (*newContrast, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setEdges (*edges, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setFillColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setFillRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color, colorSpace=None, operation="", log=None*)

Hard setter for *foreColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setForeRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOpacity (*newOpacity, operation="", log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos (*newPos, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color, operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setRadius (*radius, operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setSize (*newSize, operation=""*, *units=None, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setVertices (*value=None, operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.23 RadialStim

Attributes

<code>RadialStim(win[, tex, mask, units, pos, ...])</code>	Stimulus object for drawing radial stimuli.
<code>RadialStim.win</code>	The <i>Window</i> object in which the stimulus will be rendered by default.
<code>RadialStim.tex</code>	Texture to used on the stimulus as a grating (aka carrier)
<code>RadialStim.mask</code>	The alpha mask that forms the shape of the resulting image.
<code>RadialStim.units</code>	
<code>RadialStim.pos</code>	The position of the center of the stimulus in the stimulus <i>units</i>
<code>RadialStim.ori</code>	The orientation of the stimulus (in degrees).
<code>RadialStim.size</code>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<code>RadialStim.contrast</code>	A value that is simply multiplied by the color.
<code>RadialStim.color</code>	Alternative way of setting <i>foreColor</i> .
<code>RadialStim.colorSpace</code>	The name of the color space currently being used
<code>RadialStim.opacity</code>	Determines how visible the stimulus is relative to background.
<code>RadialStim.interpolate</code>	Whether to interpolate (linearly) the texture in the stimulus.
<code>RadialStim.setAngularCycles(value[, ...])</code>	Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message
<code>RadialStim.setAngularPhase(value[, ...])</code>	Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message
<code>RadialStim.setRadialCycles(value[, ...])</code>	Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message
<code>RadialStim.setRadialPhase(value[, ...])</code>	Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message
<code>RadialStim.name</code>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<code>RadialStim.autoLog</code>	Whether every change in this stimulus should be auto logged.
<code>RadialStim.draw([win])</code>	Draw the stimulus in its relevant window.
<code>RadialStim.autoDraw</code>	Determines whether the stimulus should be automatically drawn on every frame flip.
<code>RadialStim.clearTextures()</code>	Clear all textures associated with the stimulus.

Details

```
class psychopy.visual.RadialStim(win, tex='sqrXsqr', mask='none', units='', pos=0.0, 0.0,
    size=1.0, 1.0, radialCycles=3, angularCycles=4, radial-
    Phase=0, angularPhase=0, ori=0.0, texRes=64, angu-
    larRes=100, visibleWedge=0, 360, rgb=None, color=1.0,
    1.0, 1.0, colorSpace='rgb', dkl=None, lms=None, con-
    trast=1.0, opacity=1.0, depth=0, rgbPedestal=0.0, 0.0,
    0.0, interpolate=False, name=None, autoLog=None,
    maskParams=None)
```

Stimulus object for drawing radial stimuli.

Examples: annulus, rotating wedge, checkerboard.

Ideal for fMRI retinotopy stimuli!

Many of the capabilities are built on top of the GratingStim.

This stimulus is still relatively new and I'm finding occasional glitches. It also takes longer to draw than a typical GratingStim, so not recommended for tasks where high frame rates are needed.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

_calcPosRendered()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*.

_calcSizeRendered()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*

_createTexture(tex, id, pixFormat, stim, res=128, maskParams=None, forcePOW2=True,
 dataType=None, wrapping=True)

Create a new OpenGL 2D image texture.

Parameters

- **tex** (*Any*) – Texture data. Value can be anything that resembles image data.
- **id** (int or GLint) – Texture ID.
- **pixFormat** (GLenum or int) – Pixel format to use, values can be *GL_ALPHA* or *GL_RGB*.
- **stim** (*Any*) – Stimulus object using the texture.
- **res** (*int*) – The resolution of the texture (unless a bitmap image is used).
- **maskParams** (*dict or None*) – Additional parameters to configure the mask used with this texture.
- **forcePOW2** (*bool*) – Force the texture to be stored in a square memory area. For grating stimuli (anything that needs multiple cycles) *forcePOW2* should be set to be *True*. Otherwise the wrapping of the texture will not work.
- **dataType** (class:~pyglet.gl.*GLenum*, int or None) – None, *GL_UNSIGNED_BYTE*, *GL_FLOAT*. Only affects image files (numpy arrays will be float).
- **wrapping** (*bool*) – Enable wrapping of the texture. A texture will be set to repeat (or tile).

_getDesiredRGB(rgb, colorSpace, contrast)

Convert color to RGB while adding contrast. Requires *self.rgb*, *self.colorSpace* and *self.contrast*

`_getPolyAsRendered ()`
DEPRECATED. Return a list of vertices as rendered.

`_selectWindow (win)`
Switch drawing to the specified window. Calls the window's `_setCurrent()` method which handles the switch.

`_set (attrib, val, op="", log=None)`
DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

`_setRadialAttribute (attr, value)`
Internal helper function to reduce redundancy

`_updateEverything ()`
Internal helper function for `angularRes` and `visibleWedge` (and `init`)

`_updateList ()`
The user shouldn't need this method since it gets called after every call to `.set()` Chooses between using and not using shaders each call.

`_updateListShaders ()`
The user shouldn't need this method since it gets called after every call to `.set()` Basically it updates the OpenGL representation of your stimulus if some parameter of the stimulus changes. Call it if you change a property manually rather than using the `.set()` command

`_updateMaskCoords ()`
calculate mask coords

`_updateTextureCoords ()`
calculate texture coordinates if `angularCycles` or `Phase` change

`_updateVertices ()`
Sets `Stim.verticesPix` and `._borderPix` from `pos`, `size`, `ori`, `flipVert`, `flipHoriz`

`_updateVerticesBase ()`
Update the base vertices if angular resolution changes.

These will be multiplied by the size and rotation matrix before rendering.

property anchor

`angularCycles`
Float (but Int is prettiest). Set the number of cycles going around the stimulus. i.e. it controls the number of 'spokes'.

Operations supported.

`angularPhase`
Float. Set the angular phase (like orientation) of the texture (wraps 0-1).

This is akin to setting the orientation of the texture around the stimulus in radians. If possible, it is more efficient to rotate the stimulus using its *ori* setting instead.

Operations supported.

`angularRes`
The number of triangles used to make the sti.

Operations supported.

`autoDraw`
Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

autoLog

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property backColor

Alternative way of setting fillColor

property backColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

blendmode

The OpenGL mode in which the stimulus is draw

Can be 'avg' or 'add'. Average (avg) places the new stimulus over the old one with a transparency given by its opacity. Opaque stimuli will hide other stimuli transparent stimuli won't. Add performs the arithmetic sum of the new stimulus and the ones already present.

property borderColor

property borderColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

clearTextures ()

Clear all textures associated with the stimulus.

As of v1.61.00 this is called automatically during garbage collection of your stimulus, so doesn't need calling explicitly by the user.

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'
```

(continues on next page)

(continued from previous page)

```
# Make it light green again
stim.color = (128, 255, 128)
```

contains (*x, y=None, units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such as a *Mouse*.**

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (*pos*), and orientation (*ori*) settings (and by the vertices for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw (*win=None*)

Draw the stimulus in its relevant window. You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

If *win* is specified then override the normal window of this stimulus.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
                             # a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
                             # DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
                        # a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
                 # which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
                 # space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use `setColor` if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use `obj._foreColor.rgb`

Type DEPRECATED

property height

interpolate

Whether to interpolate (linearly) the texture in the stimulus.

If set to `False` then nearest neighbour will be used when needed, otherwise some form of interpolation will be used.

property lineColor

Alternative way of setting `borderColor`.

property lineColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use `obj._borderColor.rgb`

Type DEPRECATED

mask

The alpha mask that forms the shape of the resulting image.

Value should be one of:

- ‘circle’, ‘gauss’, ‘raisedCos’, **None** (resets to default)
- or the name of an image file (most formats supported)
- or a numpy array (1xN) ranging -1:1

Note that the mask for *RadialStim* is somewhat different to the mask for *ImageStim*. For *RadialStim* it is a 1D array specifying the luminance profile extending outwards from the center of the stimulus, rather than a 2D array

maskParams

Various types of input. Default to *None*.

This is used to pass additional parameters to the mask if those are needed.

- **For ‘gauss’ mask, pass dict {‘sd’: 5} to control** standard deviation.
- **For the ‘raisedCos’ mask, pass a dict: {‘fringeWidth’:0.2}**, where ‘fringeWidth’ is a parameter (float, 0-1), determining the proportion of the patch that will be blurred by the raised cosine edge.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = `None` your stimulus will be called “unnamed <type>”, e.g. `visual.TextStim(win)` will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, `shapeContains.py`

phase

Phase of the stimulus in each dimension of the texture.

Should be an *x,y-pair* or *scalar*

NB phase has modulus 1 (rather than 360 or 2π) This is a little unconventional but has the nice effect that setting `phase=t*n` drifts a stimulus at n Hz

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

radialCycles

Float (but Int is prettiest). Set the number of texture cycles from centre to periphery, i.e. it controls the number of ‘rings’.

Operations supported.

radialPhase

Float. Set the radial phase of the texture (wraps 0-1). This is the phase of the texture from the centre to the perimeter of the stimulus (in radians). Can be used to drift concentric rings out/inwards.

Operations supported.

setAnchor (*value, log=None*)

setAngularCycles (*value*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setAngularPhase (*value*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setAutoDraw (*value*, *log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBackColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setBackRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBlendmode (*value*, *log=None*)

DEPRECATED. Use 'stim.parameter = value' syntax instead

setBorderColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setBorderRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setContrast (*newContrast*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setDKL (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setFillColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setFillRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *foreColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setForeRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setLineRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setMask (*value*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setOpacity (*newOpacity*, *operation=""*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPhase (*value*, *operation=""*, *log=None*)

DEPRECATED. Use 'stim.parameter = value' syntax instead

setPos (*newPos*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setRadialCycles (*value*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setRadialPhase (*value*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setSF (*value*, *operation=""*, *log=None*)

DEPRECATED. Use 'stim.parameter = value' syntax instead

setSize (*newSize*, *operation=""*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setTex (*value*, *log=None*)

DEPRECATED. Use 'stim.parameter = value' syntax instead

sf

Spatial frequency of the grating texture

Should be a *x,y-pair* or *scalar* or None. If *units == 'deg'* or *'cm'* units are in cycles per deg or cm as appropriate. If *units == 'norm'* then sf units are in cycles per stimulus (and so SF scales with stimulus size). If texture is an image loaded from a file then *sf=None* defaults to *1/stimSize* to give one cycle of the image.

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

tex

Texture to used on the stimulus as a grating (aka carrier)

This can be one of various options:

- ‘sin’, ‘sqr’, ‘saw’, ‘tri’, None (resets to default)
- the name of an image file (most formats supported)
- a numpy array (1xN or NxN) ranging -1:1

If specifying your own texture using an image or numpy array you should ensure that the image has square power-of-two dimensions (e.g. 256 x 256). If not then PsychoPy will upsample your stimulus to the next larger power of two.

texRes

Power-of-two int. Sets the resolution of the mask and texture. *texRes* is overridden if an array or image is provided as mask.

Operations supported.

property units

updateColors()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

visibleWedge

tuple (start, end) in degrees. Determines visible range.

(0, 360) is full visibility.

Operations supported.

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
```

(continues on next page)

(continued from previous page)

```

        # monitor update
win2.flip() # wait for vertical blanking.

```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```

stim.draw(win1)
stim.draw(win2)

```

9.3.24 RatingScale

class psychopy.visual.**RatingScale** (*args, **kwargs)

A class for obtaining ratings, e.g., on a 1-to-7 or categorical scale.

A RatingScale instance is a re-usable visual object having a `draw()` method, with customizable appearance and response options. `draw()` displays the rating scale, handles the subject's mouse or key responses, and updates the display. When the subject accepts a selection, `.noResponse` goes `False` (i.e., there is a response).

You can call the `getRating()` method anytime to get a rating, `getRT()` to get the decision time, or `getHistory()` to obtain the entire set of (rating, RT) pairs.

There are five main elements of a rating scale: the *scale* (text above the line intended to be a reminder of how to use the scale), the *line* (with tick marks), the *marker* (a moveable visual indicator on the line), the *labels* (text below the line that label specific points), and the *accept* button. The appearance and function of elements can be customized by the experimenter; it is not possible to orient a rating scale to be vertical. Multiple scales can be displayed at the same time, and continuous real-time ratings can be obtained from the history.

The Builder RatingScale component gives a restricted set of options, but also allows full control over a RatingScale via the 'customize_everything' field.

A RatingScale instance has no idea what else is on the screen. The experimenter has to draw the item to be rated, and handle *escape* to break or quit, if desired. The subject can use the mouse or keys to respond. Direction keys (left, right) will move the marker in the smallest available increment (e.g., 1/10th of a tick-mark if precision = 10).

Example 1:

A basic 7-point scale:

```

ratingScale = visual.RatingScale(win)
item = <statement, question, image, movie, ...>
while ratingScale.noResponse:
    item.draw()
    ratingScale.draw()
    win.flip()
rating = ratingScale.getRating()
decisionTime = ratingScale.getRT()
choiceHistory = ratingScale.getHistory()

```

Example 2:

For fMRI, sometimes only a keyboard can be used. If your response box sends keys 1-4, you could specify left, right, and accept keys, and not need a mouse:

```
ratingScale = visual.RatingScale(
    win, low=1, high=5, markerStart=4,
    leftKeys='1', rightKeys = '2', acceptKeys='4')
```

Example 3:

Categorical ratings can be obtained using choices:

```
ratingScale = visual.RatingScale(
    win, choices=['agree', 'disagree'],
    markerStart=0.5, singleClick=True)
```

For other examples see Coder Demos -> stimuli -> ratingScale.py.

Authors

- 2010 Jeremy Gray: original code and on-going updates
- 2012 Henrik Singmann: tickMarks, labels, ticksAboveLine
- 2014 Jeremy Gray: multiple API changes (v1.80.00)

Parameters

win : A *Window* object (required).

choices : A list of items which the subject can choose among. *choices* takes precedence over *low*, *high*, *precision*, *scale*, *labels*, and *tickMarks*.

low : Lowest numeric rating (integer), default = 1.

high : Highest numeric rating (integer), default = 7.

precision : Portions of a tick to accept as input [1, 10, 60, 100]; default = 1 (a whole tick). Pressing a key in *leftKeys* or *rightKeys* will move the marker by one portion of a tick. *precision=60* is intended to support ratings of time-based quantities, with seconds being fractional minutes (or minutes being fractional hours). The display uses a colon (min:sec, or hours:min) to signal this to participants. The value returned by *getRating()* will be a proportion of a minute (e.g., 1:30 -> 1.5, or 59 seconds -> 59/60 = 0.98333). hours:min:sec is not supported.

scale : Optional reminder message about how to respond or rate an item, displayed above the line; default = '<low>=not at all, <high>=extremely'. To suppress the scale, set *scale=None*.

labels : Text to be placed at specific tick marks to indicate their value. Can be just the ends (if given 2 labels), ends + middle (if given 3 labels), or all points (if given the same number of labels as points).

tickMarks : List of positions at which tick marks should be placed from low to high. The default is to space tick marks equally, one per integer value.

tickHeight : The vertical height of tick marks: 1.0 is the default height (above line), -1.0 is below the line, and 0.0 suppresses the display of tickmarks. *tickHeight* is purely cosmetic, and can be fractional, e.g., 1.2.

marker : The moveable visual indicator of the current selection. The predefined styles are 'triangle', 'circle', 'glow', 'slider', and 'hover'. A slider moves smoothly when there are enough screen positions to move through, e.g., low=0, high=100. Hovering requires a set of choices, and allows clicking directly on individual choices; dwell-time is not recorded. Can also be set to a custom marker stimulus: any object with a *.draw()* method and *.pos* will work, e.g., *visual.TextStim(win, text='[]', units='norm')*.

- markerStart** : The location or value to be pre-selected upon initial display, either numeric or one of the choices. Can be fractional, e.g., midway between two options.
- markerColor** : Color to use for a predefined marker style, e.g., 'DarkRed'.
- markerExpansion** : Only affects the *glow* marker: How much to expand or contract when moving rightward; 0=none, negative shrinks.
- singleClick** : Enable a mouse click to both select and accept the rating, default = `False`. A legal key press will also count as a `singleClick`. The 'accept' box is visible, but clicking it has no effect.
- pos** [tuple (x, y)] Position of the rating scale on the screen. The midpoint of the line will be positioned at (x, y); default = (0.0, -0.4) in norm units
- size** : How much to expand or contract the overall rating scale display. Default size = 1.0. For larger than the default, set `size > 1`; for smaller, set `< 1`.
- stretch**: Like `size`, but only affects the horizontal direction.
- textSize** : The size of text elements, relative to the default size (i.e., a scaling factor, not points).
- textColor** : Color to use for labels and scale text; default = 'LightGray'.
- textFont** : Name of the font to use; default = 'Helvetica Bold'.
- showValue** : Show the subject their current selection default = `True`. Ignored if `singleClick` is `True`.
- showAccept** : Show the button to click to accept the current value by using the mouse; default = `True`.
- acceptPreText** : The text to display before any value has been selected.
- acceptText** : The text to display in the 'accept' button after a value has been selected.
- acceptSize** : The width of the accept box relative to the default (e.g., 2 is twice as wide).
- acceptKeys** : A list of keys that are used to accept the current response; default = 'return'.
- leftKeys** : A list of keys that each mean "move leftwards"; default = 'left'.
- rightKeys** : A list of keys that each mean "move rightwards"; default = 'right'.
- respKeys** : A list of keys to use for selecting choices, in the desired order. The first item will be the left-most choice, the second item will be the next choice, and so on.
- skipKeys** : List of keys the subject can use to skip a response, default = 'tab'. To require a response to every item, set `skipKeys=None`.
- lineColor** : The RGB color to use for the scale line, default = 'White'.
- mouseOnly** : Require the subject to use the mouse (any keyboard input is ignored), default = `False`. Can be used to avoid competing with other objects for keyboard input.
- noMouse**: Require the subject to use keys to respond; disable and hide the mouse. *markerStart* will default to the left end.
- minTime** : Seconds that must elapse before a response can be accepted, default = 0.4.
- maxTime** : Seconds after which a response cannot be accepted. If `maxTime <= minTime`, there's no time limit. Default = 0.0 (no time limit).
- disappear** : Whether the rating scale should vanish after a value is accepted. Can be useful when showing multiple scales.
- flipVert** : Whether to mirror-reverse the rating scale in the vertical direction.

9.3.25 psychopy.visual.Rect

Stimulus class for drawing rectangles and squares.

Overview

<code>Rect(win[, width, height, units, lineWidth, ...])</code>	Creates a rectangle of given width and height as a special case of a <code>ShapeStim</code> .
<code>Rect.width</code>	
<code>Rect.height</code>	
<code>Rect.units</code>	
<code>Rect.lineWidth</code>	Width of the line in pixels .
<code>Rect.lineColor</code>	Alternative way of setting <code>borderColor</code> .
<code>Rect.lineColorSpace</code>	Deprecated, please use <code>colorSpace</code> to set color space for the entire object
<code>Rect.fillColor</code>	Set the fill color for the shape.
<code>Rect.fillColorSpace</code>	Deprecated, please use <code>colorSpace</code> to set color space for the entire object.
<code>Rect.pos</code>	The position of the center of the stimulus in the stimulus <i>units</i>
<code>Rect.size</code>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<code>Rect.ori</code>	The orientation of the stimulus (in degrees).
<code>Rect.opacity</code>	Determines how visible the stimulus is relative to background.
<code>Rect.contrast</code>	A value that is simply multiplied by the color.
<code>Rect.depth</code>	DEPRECATED, depth is now controlled simply by drawing order.
<code>Rect.interpolate</code>	If <code>True</code> the edge of the line will be anti-aliased.
<code>Rect.lineRGB</code>	Legacy property for setting the border color of a stimulus in RGB, instead use <code>obj._borderColor.rgb</code>
<code>Rect.fillRGB</code>	Legacy property for setting the fill color of a stimulus in RGB, instead use <code>obj._fillColor.rgb</code>
<code>Rect.name</code>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<code>Rect.autoLog</code>	Whether every change in this stimulus should be auto logged.
<code>Rect.autoDraw</code>	Determines whether the stimulus should be automatically drawn on every frame flip.
<code>Rect.color</code>	Set the color of the shape.
<code>Rect.colorSpace</code>	The name of the color space currently being used

Details

```
class psychopy.visual.rect.Rect (win, width=0.5, height=0.5, units="", lineWidth=1.5, lineColor=None, lineColorSpace=None, fillColor='white', fillColorSpace=None, pos=0, 0, size=None, anchor=None, ori=0.0, opacity=None, contrast=1.0, depth=0, interpolate=True, lineRGB=False, fillRGB=False, name=None, autoLog=None, autoDraw=False, color=None, colorSpace='rgb')
```

Creates a rectangle of given width and height as a special case of a `ShapeStim`.

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **width** (*float or int*) – The width or height of the shape. *DEPRECATED* use *size* to define the dimensions of the shape on initialization. If *size* is specified the values of *width* and *height* are ignored. This is to provide legacy compatibility for existing applications.
- **height** (*float or int*) – The width or height of the shape. *DEPRECATED* use *size* to define the dimensions of the shape on initialization. If *size* is specified the values of *width* and *height* are ignored. This is to provide legacy compatibility for existing applications.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **lineWidth** (*float*) – Width of the shape’s outline.
- **lineColor** (*array_like, str, Color or None*) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **fillColor** (*array_like, str, Color or None*) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **lineColorSpace** (*str*) – Colorspace to use for the outline and fill. These change how the values passed to *lineColor* and *fillColor* are interpreted. *Deprecated*. Please use *colorSpace* to set both outline and fill colorspace. These arguments may be removed in a future version.
- **fillColorSpace** (*str*) – Colorspace to use for the outline and fill. These change how the values passed to *lineColor* and *fillColor* are interpreted. *Deprecated*. Please use *colorSpace* to set both outline and fill colorspace. These arguments may be removed in a future version.
- **pos** (*array_like*) – Initial position (*x, y*) of the shape on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is *(0.0, 0.0)* which results in no translation.
- **size** (*array_like, float, int or None*) – Width and height of the shape as *(w, h)* or *[w, h]*. If a single value is provided, the width and height will be set to the same specified value. If *None* is specified, the *size* will be set with values passed to *width* and *height*.
- **ori** (*float*) – Initial orientation of the shape in degrees about its origin. Positive values will rotate the shape clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being

blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.

- **contrast** (*float*) – Contrast level of the shape (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the shape when *autoDraw* is enabled. *DEPRECATED*
- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing shape outlines. This produces a smoother (less-pixelated) outline of the shape.
- **lineRGB** (array_like, *Color* or None) – *Deprecated*. Please use *lineColor* and *fillColor*. These arguments may be removed in a future version.
- **fillRGB** (array_like, *Color* or None) – *Deprecated*. Please use *lineColor* and *fillColor*. These arguments may be removed in a future version.
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the *draw()* method.
- **color** (array_like, str, *Color* or None) – Sets both the initial *lineColor* and *fillColor* of the shape.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.

width, height

The width and height of the rectangle. Values are aliased with fields in the *size* attribute. Use these values to adjust the size of the rectangle in a single dimension after initialization.

Type float or int

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

static _calcEquilateralVertices (*edges, radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

_calcPosRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*.

_calcSizeRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by *_updateVertices()* and *verticesPix*

_getDesiredRGB (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires *self.rgb*, *self.colorSpace* and *self.contrast*

_getPolyAsRendered ()

DEPRECATED. Return a list of vertices as rendered.

_selectWindow (*win*)

Switch drawing to the specified window. Calls the window's *_setCurrent()* method which handles the switch.

_set (*attrib, val, op="", log=None*)

DEPRECATED since 1.80.04 + 1. Use *setAttribute()* and *val2array()* instead.

`_updateList ()`

The user shouldn't need this method since it gets called after every call to `.set()` Chooses between using and not using shaders each call.

`_updateVertices ()`

Sets `Stim.verticesPix` and `._borderPix` from `pos`, `size`, `ori`, `flipVert`, `flipHoriz`

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property `backColor`

Alternative way of setting `fillColor`

property `backColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object.

property `backRGB`

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property `borderColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object

property `borderRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use `obj._borderColor.rgb`

Type DEPRECATED

`closeShape`

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, `closeShape=True` is assumed and shouldn't be changed.

`color`

Set the color of the shape. Sets both `fillColor` and `lineColor` simultaneously if applicable.

property `colorSpace`

The name of the color space currently being used

Value should be: a string or `None`

For strings and hex values this is not needed. If `None` the default `colorSpace` for the stimulus is used (defined during initialisation).

Please note that changing `colorSpace` does not change stimulus parameters. Thus you usually want to specify `colorSpace` before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'
```

(continues on next page)

(continued from previous page)

```
# Make it light green again
stim.color = (128, 255, 128)
```

contains (*x, y=None, units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such as a *Mouse*.**

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (*pos*), and orientation (*ori*) settings (and by the vertices for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw (*win=None, keepMatrix=False*)

Draw the stimulus in its relevant window.

You must call this method after every `MyWin.flip()` if you want the stimulus to appear on that frame and then update the screen again.

property fillColor

Set the fill color for the shape.

property fillColorSpaceDeprecated, please use `colorSpace` to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use `setColor` if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

interpolate

If *True* the edge of the line will be anti-aliased.

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

lineWidth

Width of the line in **pixels**.

Operations supported.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. *visual.TextStim(win)* will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, *shapeContains.py*

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:


```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAutoDraw (*value*, *log=None*)

Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setBackRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setBorderRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Sets both the line and fill to be the same color.

setContrast (*newContrast*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setDKL (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setFillColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setFillRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *foreColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setForeRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setHeight (*height*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setLMS (*color, operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOpacity (*newOpacity, operation="", log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos (*newPos, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize (*size, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

Operations supported.

setVertices (*value=None, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setWidth (*width, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.26 psychopy.visual.Rift

Overview

<i>Rift</i> ([fovType, trackingOriginType, ...])	Class provides a display and peripheral interface for the Oculus Rift (see: https://www.oculus.com/) head-mounted display.
<i>Rift.close</i> ()	Close the window and cleanly shutdown the LibOVR session.
<i>Rift.size</i>	Size property to get the dimensions of the view buffer instead of the window.
<i>Rift.setSize</i> (value[, log])	
<i>Rift.perfHudMode</i> ([mode])	Set the performance HUD mode.
<i>Rift.hidePerfHud</i> ()	Hide the performance HUD.
<i>Rift.stereoDebugHudMode</i> (mode)	Set the debug stereo HUD mode.
<i>Rift.setStereoDebugHudOption</i> (option, value)	Configure stereo debug HUD guides.
<i>Rift.userHeight</i>	Get user height in meters (<i>float</i>).
<i>Rift.eyeHeight</i>	Eye height in meters (<i>float</i>).
<i>Rift.eyeToNoseDistance</i>	Eye to nose distance in meters (<i>float</i>).
<i>Rift.eyeOffset</i>	Eye separation in centimeters (<i>float</i>).
<i>Rift.hasPositionTracking</i>	True if the HMD is capable of tracking position.
<i>Rift.hasOrientationTracking</i>	True if the HMD is capable of tracking orientation.
<i>Rift.hasMagYawCorrection</i>	True if this HMD supports yaw drift correction.
<i>Rift.manufacturer</i>	Get the connected HMD's manufacturer (<i>str</i>).
<i>Rift.serialNumber</i>	Get the connected HMD's unique serial number (<i>str</i>).
<i>Rift.hid</i>	USB human interface device (HID) identifiers (<i>int, int</i>).
<i>Rift.displayResolution</i>	Get the HMD's raster display size (<i>int, int</i>).
<i>Rift.displayRefreshRate</i>	Get the HMD's display refresh rate in Hz (<i>float</i>).
<i>Rift.pixelsPerTanAngleAtCenter</i>	Horizontal and vertical pixels per tangent angle (=1) at the center of the display.

continues on next page

Table 9.19 – continued from previous page

<code>Rift.tanAngleToNDC(horzTan, vertTan)</code>	Convert tan angles to the normalized device coordinates for the current buffer.
<code>Rift.trackerCount</code>	Number of attached trackers.
<code>Rift.getTrackerInfo(trackerIdx)</code>	Get tracker information.
<code>Rift.headLocked</code>	<i>True</i> if head locking is enabled.
<code>Rift.trackingOriginType</code>	Current tracking origin type (<i>str</i>).
<code>Rift.recenterTrackingOrigin()</code>	Recenter the tracking origin using the current head position.
<code>Rift.specifyTrackingOrigin(pose)</code>	Specify a tracking origin.
<code>Rift.specifyTrackingOriginPosOri([pos, ori])</code>	Specify a tracking origin using a pose and orientation.
<code>Rift.clearShouldRecenterFlag()</code>	Clear the ‘shouldRecenter’ status flag at the API level.
<code>Rift.testBoundary(deviceType[, bounadry-Type])</code>	Test if tracked devices are colliding with the play area boundary.
<code>Rift.sensorSampleTime</code>	Sensor sample time (<i>float</i>).
<code>Rift.getDevicePose(deviceName[, absTime, ...])</code>	Get the pose of a tracked device.
<code>Rift.getTrackingState([absTime, latency-Marker])</code>	Get the tracking state of the head and hands.
<code>Rift.calcEyePoses(headPose[, originPose])</code>	Calculate eye poses for rendering.
<code>Rift.eyeRenderPose</code>	Computed eye pose for the current buffer.
<code>Rift.shouldQuit</code>	<i>True</i> if the user requested the application should quit through the headset’s interface.
<code>Rift.isVisible</code>	<i>True</i> if the app has focus in the HMD and is visible to the viewer.
<code>Rift.hmdMounted</code>	<i>True</i> if the HMD is mounted on the user’s head.
<code>Rift.hmdPresent</code>	<i>True</i> if the HMD is present.
<code>Rift.shouldRecenter</code>	<i>True</i> if the user requested the origin be re-centered through the headset’s interface.
<code>Rift.hasInputFocus</code>	<i>True</i> if the application currently has input focus.
<code>Rift.overlayPresent</code>	
<code>Rift.setBuffer(buffer[, clear])</code>	Set the active draw buffer.
<code>Rift.getPredictedDisplayTime()</code>	Get the predicted time the next frame will be displayed on the HMD.
<code>Rift.getTimeInSeconds()</code>	Absolute time in seconds.
<code>Rift.viewMatrix</code>	The view matrix for the current eye buffer.
<code>Rift.nearClip</code>	Distance to the near clipping plane in meters.
<code>Rift.farClip</code>	Distance to the far clipping plane in meters.
<code>Rift.projectionMatrix</code>	Get the projection matrix for the current eye buffer.
<code>Rift.isBoundaryVisible</code>	<i>True</i> if the VR boundary is visible.
<code>Rift.getBoundaryDimensions([boundaryType])</code>	Get boundary dimensions.
<code>Rift.connectedControllers</code>	Connected controller types (<i>list of str</i>)
<code>Rift.updateInputState([controllers])</code>	Update all connected controller states.
<code>Rift.flip([clearBuffer, drawMirror])</code>	Submit view buffer images to the HMD’s compositor for display at next V-SYNC and draw the mirror texture to the on-screen window.
<code>Rift.multiplyViewMatrixGL()</code>	Multiply the local eye pose transformation modelMatrix obtained from the SDK using <code>glMultMatrixf</code> .
<code>Rift.multiplyProjectionMatrixGL()</code>	Multiply the current projection modelMatrix obtained from the SDK using <code>glMultMatrixf</code> .
<code>Rift.setRiftView([clearDepth])</code>	Set head-mounted display view.

continues on next page

Table 9.19 – continued from previous page

<code>Rift.setDefaultView([clearDepth])</code>	Return to default projection.
<code>Rift.getThumbstickValues([controller, dead-zone])</code>	Get controller thumbstick values.
<code>Rift.getIndexTriggerValues([controller, ...])</code>	Get the values of the index triggers.
<code>Rift.getHandTriggerValues([controller, dead-zone])</code>	Get the values of the hand triggers.
<code>Rift.getButtons(buttons[, controller, testState])</code>	Get button states from a controller.
<code>Rift.getTouches(touches[, controller, testState])</code>	Get touch states from a controller.
<code>Rift.startHaptics(controller[, frequency, ...])</code>	Start haptic feedback (vibration).
<code>Rift.stopHaptics(controller)</code>	Stop haptic feedback.
<code>Rift.createHapticsBuffer(samples)</code>	Create a new haptics buffer.
<code>Rift.submitControllerVibration(controller, ...)</code>	Submit a haptics buffer to begin controller vibration.
<code>Rift.createPose([pos, ori])</code>	Create a new Rift pose object (<code>LibOVRPose</code>).
<code>Rift.createBoundingBox([extents])</code>	Create a new bounding box object (<code>LibOVRBounds</code>).
<code>Rift.isPoseVisible(pose)</code>	Check if a pose object is visible to the present eye.

Details

class `psychopy.visual.rift.Rift` (*fovType*='recommended', *trackingOriginType*='floor', *texelsPerPixel*=1.0, *headLocked*=False, *highQuality*=True, *monoscopic*=False, *samples*=1, *mirrorMode*='default', *mirrorRes*=None, *warnAppFrameDropped*=True, *autoUpdateInput*=True, *legacyOpenGL*=True, *args, **kwargs)

Class provides a display and peripheral interface for the Oculus Rift (see: <https://www.oculus.com/>) head-mounted display.

Requires PsychXR 0.2.4 to be installed. Setting the *winType*='glfw' is preferred for VR applications.

Parameters

- **fovType** (*str*) – Field-of-view (FOV) configuration type. Using 'recommended' auto-configures the FOV using the recommended parameters computed by the runtime. Using 'symmetric' forces a symmetric FOV using optimal parameters from the SDK, this mode is required for displaying 2D stimuli. Specifying 'max' will use the maximum FOVs supported by the HMD.
- **trackingOriginType** (*str*) – Specify the HMD origin type. If 'floor', the height of the user is added to the head tracker by LibOVR.
- **texelsPerPixel** (*float*) – Texture pixels per display pixel at FOV center. A value of 1.0 results in 1:1 mapping. A fractional value results in a lower resolution draw buffer which may increase performance.
- **headLocked** (*bool*) – Lock the compositor render layer in-place, disabling Asynchronous Space Warp (ASW). Enable this if you plan on computing eye poses using custom or modified head poses.
- **highQuality** (*bool*) – Configure the compositor to use anisotropic texture sampling (4x). This reduces aliasing artifacts resulting from high frequency details particularly in the periphery.
- **nearClip** (*float*) – Location of the near and far clipping plane in GL units (meters by default) from the viewer. These values can be updated after initialization.

- **farClip** (*float*) – Location of the near and far clipping plane in GL units (meters by default) from the viewer. These values can be updated after initialization.
- **monoscopic** (*bool*) – Enable monoscopic rendering mode which presents the same image to both eyes. Eye poses used will be both centered at the HMD origin. Monoscopic mode uses a separate rendering pipeline which reduces VRAM usage. When in monoscopic mode, you do not need to call ‘setBuffer’ prior to rendering (doing so will do have no effect).
- **samples** (*int or str*) – Specify the number of samples for multi-sample anti-aliasing (MSAA). When >1, multi-sampling logic is enabled in the rendering pipeline. If ‘max’ is specified, the largest number of samples supported by the platform is used. If floating point textures are used, MSAA sampling is disabled. Must be power of two value.
- **mirrorMode** (*str*) – On-screen mirror mode. Values ‘left’ and ‘right’ show rectilinear images of a single eye. Value ‘distortion’ shows the post-distortion image after being processed by the compositor. Value ‘default’ displays rectilinear images of both eyes side-by-side.
- **mirrorRes** (*list of int*) – Resolution of the mirror texture. If *None*, the resolution will match the window size. The value of *mirrorRes* is used for to define the resolution of movie frames.
- **warnAppFrameDropped** (*bool*) – Log a warning if the application drops a frame. This occurs when the application fails to submit a frame to the compositor on-time. Application frame drops can have many causes, such as running routines in your application loop that take too long to complete. However, frame drops can happen sporadically due to driver bugs and running background processes (such as Windows Update). Use the performance HUD to help diagnose the causes of frame drops.
- **autoUpdateInput** (*bool*) – Automatically update controller input states at the start of each frame. If *False*, you must manually call *updateInputState* before getting input values from *LibOVR* managed input devices.
- **legacyOpenGL** (*bool*) – Disable ‘immediate mode’ OpenGL calls in the rendering pipeline. Specifying *False* maintains compatibility with existing PsychoPy stimuli drawing routines. Use *True* when computing transformations using some other method and supplying shaders matrices directly.

_assignFlipTime (*obj, attrib*)

Helper function to assign the time of last flip to the obj.attrib

Parameters

- **obj** (*dict or object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of *obj* to assign the flip time to.

_checkMatchingSizes (*requested, actual*)

Checks whether the requested and actual screen sizes differ. If not then a warning is output and the window size is set to actual

_cleanEditables ()

Make sure there are no dead refs in the editables list

_endOffFlip (*clearBuffer*)

Override end of flip with custom color channel masking if required.

_getFrame (*rect=None, buffer='mirror'*)

Return the current HMD view as an image.

Parameters

- **rect** (*array_like*) – Rectangle [x, y, w, h] defining a sub-region of the frame to capture. This should remain *None* for HMD frames.
- **buffer** (*str, optional*) – Buffer to capture. For the HMD, only ‘mirror’ is available at this time.

Returns Buffer pixel contents as a PIL/Pillow image object.

Return type Image

`__getRegionOfFrame` (*rect=-1, 1, 1, -1, buffer='front', power2=False, squarePower2=False*)

Deprecated function, here for historical reasons. You may now use `:py:attr: ~Window._getFrame()` and specify a `rect` to get a sub-region, just as used here.

`power2` can be useful with older OpenGL versions to avoid interpolation in `PatchStim`. If `power2` or `squarePower2`, it will expand `rect` dimensions up to next power of two. `squarePower2` uses the max dimensions. You need to check what your hardware & OpenGL supports, and call `__getRegionOfFrame()` as appropriate.

`__prepareMonoFrame` (*clear=True*)

Prepare a frame for monoscopic rendering. This is called automatically after `__startHmdFrame()` if monoscopic rendering is enabled.

`__renderFBO` ()

Perform a warp operation.

(in this case a copy operation without any warping)

`__resolveMSAA` ()

Resolve multisample anti-aliasing (MSAA). If MSAA is enabled, drawing operations are diverted to a special multisample render buffer. Pixel data must be ‘resolved’ by blitting it to the swap chain texture. If not, the texture will be blank.

Notes

You cannot perform operations on the default FBO (at `frameBuffer`) when MSAA is enabled. Any changes will be over-written when ‘flip’ is called.

`__setCurrent` ()

Make this window’s OpenGL context current.

If called on a window whose context is current, the function will return immediately. This reduces the number of redundant calls if no context switch is required. If `useFBO=True`, the framebuffer is bound after the context switch.

`__setupFrameBuffer` ()

Override the default framebuffer init code in `window.Window` to use the HMD swap chain. The HMD’s swap texture and render buffer are configured here.

If multisample anti-aliasing (MSAA) is enabled, a secondary render buffer is created. Rendering is diverted to the multi-sample buffer when drawing, which is then resolved into the HMD’s swap chain texture prior to committing it to the chain. Consequently, you cannot pass the texture attached to the FBO specified by `frameBuffer` until the MSAA buffer is resolved. Doing so will result in a blank texture.

`__setupGL` ()

Setup OpenGL state for this window.

`__setupGamma` (*gammaVal*)

A private method to work out how to handle gamma for this Window given that the user might have specified an explicit value, or maybe gave a Monitor.

`_startHmdFrame()`

Prepare to render an HMD frame. This must be called every frame before flipping or setting the view buffer.

This function will wait until the HMD is ready to begin rendering before continuing. The current frame texture from the swap chain are pulled from the SDK and made available for binding.

`_startOfFlip()`

Custom `_startOfFlip` for HMD rendering. This finalizes the HMD texture before diverting drawing operations back to the on-screen window. This allows `flip` to swap the on-screen and HMD buffers when called. This function always returns `True`.

Returns

Return type `True`

`_updatePerfStats()`

Update and process performance statistics obtained from LibOVR. This should be called at the beginning of each frame to get the stats of the last frame.

This is called automatically when `_waitToBeginHmdFrame()` is called at the beginning of each frame.

`_updateProjectionMatrix()`

Update or re-calculate projection matrices based on the current render descriptor configuration.

`_waitToBeginHmdFrame()`

Wait until the HMD surfaces are available for rendering.

`addEditable(editable)`

Adds an editable element to the screen (something to which characters can be sent with meaning from the keyboard).

The current editable object receiving chars is `Window.currentEditable`

Parameters `editable` –

Returns

property `ambientLight`

Ambient light color for the scene [r, g, b, a]. Values range from 0.0 to 1.0. Only applicable if `useLights` is `True`.

Examples

Setting the ambient light color:

```
win.ambientLight = [0.5, 0.5, 0.5]

# don't do this!!!
win.ambientLight[0] = 0.5
win.ambientLight[1] = 0.5
win.ambientLight[2] = 0.5
```

`applyEyeTransform(clearDepth=True)`

Apply the current view and projection matrices.

Matrices specified by attributes `viewMatrix` and `projectionMatrix` are applied using ‘immediate mode’ OpenGL functions. Subsequent drawing operations will be affected until `flip()` is called.

All transformations in `GL_PROJECTION` and `GL_MODELVIEW` matrix stacks will be cleared (set to identity) prior to applying.

Parameters `clearDepth` (*bool*) – Clear the depth buffer. This may be required prior to rendering 3D objects.

Examples

Using a custom view and projection matrix:

```
# Must be called every frame since these values are reset after
# `flip()` is called!
win.viewMatrix = viewtools.lookAt( ... )
win.projectionMatrix = viewtools.perspectiveProjectionMatrix( ... )
win.applyEyeTransform()
# draw 3D objects here ...
```

property aspect

Aspect ratio of the current viewport (width / height).

blendMode

Blend mode to use.

calcEyePoses (headPose, originPose=None)

Calculate eye poses for rendering.

This function calculates the eye poses to define the viewpoint transformation for each eye buffer. Upon starting a new frame, the application loop is halted until this function is called and returns.

Once this function returns, `setBuffer` may be called and frame rendering can commence. The computed eye pose for the selected buffer is accessible through the `eyeRenderPose` attribute after calling `setBuffer()`. If `monoscopic=True`, the eye poses are set to the head pose.

The source data specified to `headPose` can originate from the tracking state retrieved by calling `getTrackingState()`, or from other sources. If a custom head pose is specified (for instance, from a motion tracker), you must ensure `head-locking` is enabled to prevent the ASW feature of the compositor from engaging. Furthermore, you must specify sensor sample time for motion-to-photon calculation derived from the sample time of the custom tracking source.

Parameters

- **headPose** (*LibOVRPose*) – Head pose to use.
- **originPose** (*LibOVRPose, optional*) – Origin of tracking in the VR scene.

Examples

Get the tracking state and calculate the eye poses:

```
# get tracking state at predicted mid-frame time
trackingState = hmd.getTrackingState()

# get the head pose from the tracking state
headPose = trackingState.headPose.thePose
hmd.calcEyePoses(headPose) # compute eye poses

# begin rendering to each eye
for eye in ('left', 'right'):
```

(continues on next page)

(continued from previous page)

```
hmd.setBuffer(eye)
hmd.setRiftView()
# draw stuff here ...
```

Using a custom head pose (make sure `headLocked=True` before doing this):

```
headPose = createPose((0., 1.75, 0.))
hmd.calcEyePoses(headPose) # compute eye poses
```

callOnFlip (*function*, *args, **kwargs)

Call a function immediately after the next `flip()` command.

The first argument should be the function to call, the following args should be used exactly as you would for your normal call to the function (can use ordered arguments or keyword arguments as normal).

e.g. If you have a function that you would normally call like this:

```
pingMyDevice(portToPing, channel=2, level=0)
```

then you could call `callOnFlip()` to have the function call synchronized with the frame flip like this:

```
win.callOnFlip(pingMyDevice, portToPing, channel=2, level=0)
```

clearBuffer (*color=True*, *depth=False*, *stencil=False*)

Clear the present buffer (to which you are currently drawing) without flipping the window.

Useful if you want to generate movie sequences from the back buffer without actually taking the time to flip the window.

Set *color* prior to clearing to set the color to clear the color buffer to. By default, the depth buffer is cleared to a value of 1.0.

Parameters

- **color** (*bool*) – Buffers to clear.
- **depth** (*bool*) – Buffers to clear.
- **stencil** (*bool*) – Buffers to clear.

Examples

Clear the color buffer to a specified color:

```
win.color = (1, 0, 0)
win.clearBuffer(color=True)
```

Clear only the depth buffer, *depthMask* must be *True* or else this will have no effect. Depth mask is usually *True* by default, but may change:

```
win.depthMask = True
win.clearBuffer(color=False, depth=True, stencil=False)
```

clearShouldRecenterFlag ()

Clear the ‘shouldRecenter’ status flag at the API level.

close ()

Close the window and cleanly shutdown the LibOVR session.

property color

Set the color of the window.

This command sets the color that the blank screen will have on the next clear operation. As a result it effectively takes TWO `flip()` operations to become visible (the first uses the color to create the new screen, the second presents that screen to the viewer). For this reason, if you want to changed background color of the window “on the fly”, it might be a better idea to draw a `Rect` that fills the whole window with the desired `Rect.fillColor` attribute. That’ll show up on first flip.

See other stimuli (e.g. `GratingStim.color`) for more info on the color attribute which essentially works the same on all PsychoPy stimuli.

See *Color spaces* for further information about the ways to specify colors and their various implications.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property connectedControllers

Connected controller types (*list of str*)

property contentScaleFactor

Scaling factor (*float*) to use when drawing to the backbuffer to convert framebuffer to client coordinates.

See also:

getContentScaleFactor

property convergeOffset

Convergence offset from monitor in centimeters.

This is value corresponds to the offset from screen plane to set the convergence plane (or point for *toe-in* projections). Positive offsets move the plane farther away from the viewer, while negative offsets nearer. This value is used by *setPerspectiveView* and should be set before calling it to take effect.

Notes

- This value is only applicable for *setToeIn* and *setOffAxisView*.

coordToRay (*screenXY*)

Convert a screen coordinate to a direction vector.

Takes a screen/window coordinate and computes a vector which projects a ray from the viewpoint through it (line-of-sight). Any 3D point touching the ray will appear at the screen coordinate.

Uses the current *viewport* and *projectionMatrix* to calculate the vector. The vector is in eye-space, where the origin of the scene is centered at the viewpoint and the forward direction aligned with the -Z axis. A ray of (0, 0, -1) results from a point at the very center of the screen assuming symmetric frustums.

Note that if you are using a flipped/mirrored view, you must invert your supplied screen coordinates (*screenXY*) prior to passing them to this function.

Parameters *screenXY* (*array_like*) – X, Y screen coordinate. Must be in units of the window.

Returns Normalized direction vector [x, y, z].

Return type ndarray

Examples

Getting the direction vector between the mouse cursor and the eye:

```
mx, my = mouse.getPos()
dir = win.coordToRay((mx, my))
```

Set the position of a 3D stimulus object using the mouse, constrained to a plane. The object origin will always be at the screen coordinate of the mouse cursor:

```
# the eye position in the scene is defined by a rigid body pose
win.viewMatrix = camera.getViewMatrix()
win.applyEyeTransform()

# get the mouse location and calculate the intercept
mx, my = mouse.getPos()
ray = win.coordToRay([mx, my])
result = intersectRayPlane( # from mathtools
    orig=camera.pos,
    dir=camera.transformNormal(ray),
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))

# if result is `None`, there is no intercept
if result is not None:
    pos, dist = result
    objModel.thePose.pos = pos
else:
    objModel.thePose.pos = (0, 0, -10) # plane origin
```

If you don't define the position of the viewer with a *RigidBodyPose*, you can obtain the appropriate eye position and rotate the ray by doing the following:

```

pos = numpy.linalg.inv(win.viewMatrix)[:3, 3]
ray = win.coordToRay([mx, my]).dot(win.viewMatrix[:3, :3])
# then ...
result = intersectRayPlane(
    orig=pos,
    dir=ray,
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))
    
```

static createBoundingBox (*extents=None*)

Create a new bounding box object (LibOVRBounds).

LibOVRBounds represents an axis-aligned bounding box with dimensions defined by *extents*. Bounding boxes are primarily used for visibility testing and culling by *PsychXR*. The dimensions of the bounding box can be specified explicitly, or fitted to meshes by passing vertices to the `fit()` method after initialization.

This function exposes the LibOVRBounds class so you don't need to access it by importing *psychxr*.

Parameters *extents* (*array_like* or *None*) – Extents of the bounding box as (*mins*, *maxs*). Where *mins* (x, y, z) is the minimum and *maxs* (x, y, z) is the maximum extents of the bounding box in world units. If *None* is specified, the returned bounding box will be invalid. The bounding box can be later constructed using the `fit()` method or the *extents* attribute.

Returns Object representing a bounding box.

Return type `~psychxr.libovr.LibOVRBounds`

Examples

Add a bounding box to a pose:

```

# create a 1 meter cube bounding box centered with the pose
bbox = Rift.createBoundingBox((-0.5, -0.5, -0.5), (0.5, 0.5, 0.5))

# create a pose and attach the bounding box
modelPose = Rift.createPose()
modelPose.boundingBox = bbox
    
```

Perform visibility culling on the pose using the bounding box by using the `isVisible()` method:

```

if hmd.isPoseVisible(modelPose):
    modelPose.draw()
    
```

static createHapticsBuffer (*samples*)

Create a new haptics buffer.

A haptics buffer is object which stores vibration amplitude samples for playback through the Touch controllers. To play a haptics buffer, pass it to `submitHapticsBuffer()`.

Parameters *samples* (*array_like*) – 1-D array of amplitude samples, ranging from 0 to 1. Values outside of this range will be clipped. The buffer must not exceed `HAPTICS_BUFFER_SAMPLES_MAX` samples, any additional samples will be dropped.

Returns Haptics buffer object.

Return type LibOVRHapticsBuffer

Notes

Methods *startHaptics* and *stopHaptics* cannot be used interchangeably with this function.

Examples

Create a haptics buffer where vibration amplitude ramps down over the course of playback:

```
samples = np.linspace(
    1.0, 0.0, num=HAPTICS_BUFFER_SAMPLES_MAX-1, dtype=np.float32)
hbuff = Rift.createHapticsBuffer(samples)

# vibrate right Touch controller
hmd.submitControllerVibration(CONTROLLER_TYPE_RTOUCH, hbuff)
```

static createPose (*pos=0.0, 0.0, 0.0, ori=0.0, 0.0, 0.0, 1.0*)

Create a new Rift pose object (LibOVRPose).

LibOVRPose is used to represent a rigid body pose mainly for use with the PsychXR's LibOVR module. There are several methods associated with the object to manipulate the pose.

This function exposes the LibOVRPose class so you don't need to access it by importing *psychxr*.

Parameters

- **pos** (*tuple, list, or ndarray of float*) – Position vector/coordinate (x, y, z).
- **ori** (*tuple, list, or ndarray of float*) – Orientation quaternion (x, y, z, w).

Returns Object representing a rigid body pose for use with LibOVR.

Return type LibOVRPose

property cullFace

True if face culling is enabled.

property cullFaceMode

Face culling mode, either *back*, *front* or *both*.

property currentEditable

The editable (Text?) object that currently has key focus

property depthFunc

Depth test comparison function for rendering.

property depthMask

True if depth masking is enabled. Writing to the depth buffer will be disabled.

property depthTest

True if depth testing is enabled.

classmethod dispatchAllWindowsEvents ()

Dispatches events for all pyglet windows. Used by iohub 2.0 psychopy kb event integration.

property displayRefreshRate

Get the HMD's display refresh rate in Hz (*float*).

property displayResolution

Get the HMD's raster display size (*int, int*).

property draw3d

True if 3D drawing is enabled on this window.

property eyeHeight

Eye height in meters (*float*).

property eyeOffset

Eye separation in centimeters (*float*).

property eyeRenderPose

Computed eye pose for the current buffer. Only valid after calling `calcEyePoses()`.

property eyeToNoseDistance

Eye to nose distance in meters (*float*).

Examples

Generate your own eye poses. These are used when `calcEyePoses()` is called:

```
leftEyePose = Rift.createPose((-self.eyeToNoseDistance, 0., 0.))
rightEyePose = Rift.createPose((self.eyeToNoseDistance, 0., 0.))
```

Get the inter-axial separation (IAS) reported by *LibOVR*:

```
iad = self.eyeToNoseDistance * 2.0
```

property farClip

Distance to the far clipping plane in meters.

property firmwareVersion

Get the firmware version of the active HMD (*int, int*).

flip (*clearBuffer=True, drawMirror=True*)

Submit view buffer images to the HMD's compositor for display at next V-SYNC and draw the mirror texture to the on-screen window. This must be called every frame.

Parameters

- **clearBuffer** (*bool*) – Clear the frame after flipping.
- **drawMirror** (*bool*) – Draw the HMD mirror texture from the compositor to the window.

Returns Absolute time in seconds when control was given back to the application. The difference between the current and previous values should be very close to 1 / refreshRate of the HMD.

Return type *float*

Notes

- The HMD compositor and application are asynchronous, therefore there is no guarantee that the timestamp returned by 'flip' corresponds to the exact vertical retrace time of the HMD.

fps ()

Report the frames per second since the last call to this function (or since the window was created if this is first call)

property framebufferSize

Size of the framebuffer in pixels (w, h).

property frontFace

Face winding order to define front, either *ccw* or *cw*.

fullscr

Set whether fullscreen mode is *True* or *False* (not all backends can toggle an open window).

gamma

Set the monitor gamma for linearization.

Warning: Don't use this if using a Bits++ or Bits#, as it overrides monitor settings.

gammaRamp

Sets the hardware CLUT using a specified 3xN array of floats ranging between 0.0 and 1.0.

Array must have a number of rows equal to $2^{\max(\text{bpc})}$.

getActualFrameRate (*nIdentical=10, nMaxFrames=100, nWarmUpFrames=10, threshold=1*)

Measures the actual frames-per-second (FPS) for the screen.

This is done by waiting (for a max of *nMaxFrames*) until *nIdentical* frames in a row have identical frame times (std dev below *threshold* ms).

Parameters

- **nIdentical** (*int, optional*) – The number of consecutive frames that will be evaluated. Higher → greater precision. Lower → faster.
- **nMaxFrames** (*int, optional*) – The maximum number of frames to wait for a matching set of *nIdentical*.
- **nWarmUpFrames** (*int, optional*) – The number of frames to display before starting the test (this is in place to allow the system to settle after opening the *Window* for the first time).
- **threshold** (*int or float, optional*) – The threshold for the std deviation (in ms) before the set are considered a match.

Returns Frame rate (FPS) in seconds. If there is no such sequence of identical frames a warning is logged and *None* will be returned.

Return type *float* or *None*

getBoundaryDimensions (*boundaryType='PlayArea'*)

Get boundary dimensions.

Parameters **boundaryType** (*str*) – Boundary type, can be 'PlayArea' or 'Outer'.

Returns Dimensions of the boundary meters [x, y, z].

Return type ndarray

getButtons (*buttons*, *controller*='Xbox', *testState*='continuous')

Get button states from a controller.

Returns *True* if any names specified to *buttons* reflect *testState* since the last *updateInputState* or *flip* call. If multiple button names are specified as a *list* or *tuple* to *buttons*, multiple button states are tested, returning *True* if all the buttons presently satisfy the *testState*. Note that not all controllers available share the same buttons. If a button is not available, this function will always return *False*.

Parameters

- **buttons** (*list* of *str* or *str*) – Buttons to test. Valid *buttons* names are 'A', 'B', 'RThumb', 'RShoulder', 'X', 'Y', 'LThumb', 'LShoulder', 'Up', 'Down', 'Left', 'Right', 'Enter', 'Back', 'VolUp', 'VolDown', and 'Home'. Names can be passed as a *list* to test multiple button states.
- **controller** (*str*) – Controller name.
- **testState** (*str*) – State to test. Valid values are:
 - **continuous** - Button is presently being held down.
 - **rising** or **pressed** - Button has been *pressed* since the last update.
 - **falling** or **released** - Button has been *released* since the last update.

Returns Button state and timestamp in seconds the controller was polled.

Return type tuple of bool, float

Examples

Check if the 'Enter' button on the Oculus remote was released:

```
isPressed, tsec = hmd.getButtons(['Enter'], 'Remote', 'falling')
```

Check if the 'A' button was pressed on the touch controller:

```
isPressed, tsec = hmd.getButtons(['A'], 'Touch', 'pressed')
```

getContentScaleFactor ()

Get the scaling factor required for scaling correctly on high-DPI displays.

If the returned value is 1.0, no scaling needs to be applied to objects drawn on the backbuffer. A value >1.0 indicates that the backbuffer is larger than the reported client area, requiring points to be scaled to maintain constant size across similarly sized displays. In other words, the scaling required to convert framebuffer to client coordinates.

Returns Scaling factor to be applied along both horizontal and vertical dimensions.

Return type float

Examples

Get the size of the client area:

```
clientSize = win.frameBufferSize / win.getContentScaleFactor()
```

Get the framebuffer size from the client size:

```
frameBufferSize = win.clientSize * win.getContentScaleFactor()
```

Convert client (window) to framebuffer pixel coordinates (eg., a mouse coordinate, vertices, etc.):

```
# `mousePosXY` is an array ...
frameBufferXY = mousePosXY * win.getContentScaleFactor()
# you can also use the attribute ...
frameBufferXY = mousePosXY * win.contentScaleFactor
```

Notes

- This value is only valid after the window has been fully realized.

getDevicePose (*deviceName*, *absTime=None*, *latencyMarker=False*)

Get the pose of a tracked device. For head (HMD) and hand poses (Touch controllers) it is better to use *getTrackingState()* instead.

Parameters

- **deviceName** (*str*) – Name of the device. Valid device names are: ‘HMD’, ‘LTouch’, ‘RTouch’, ‘Touch’, ‘Object0’, ‘Object1’, ‘Object2’, and ‘Object3’.
- **absTime** (*float, optional*) – Absolute time in seconds the device pose refers to. If not specified, the predicted time is used.
- **latencyMarker** (*bool*) – Insert a marker for motion-to-photon latency calculation. Should only be *True* if the HMD pose is being used to compute eye poses.

Returns Pose state object. *None* if device tracking was lost.

Return type *LibOVRPoseState* or *None*

getFutureFlipTime (*targetTime=0*, *clock=None*)

The expected time of the next screen refresh. This is currently calculated as *win._lastFrameTime + refreshInterval*

Parameters

- **targetTime** (*float*) – The delay *from now* for which you want the flip time. 0 will give the because that the earliest we can achieve. 0.15 will give the schedule flip time that gets as close to 150 ms as possible
- **clock** (*None, 'ptb', 'now' or any Clock object*) – If *True* then the time returned is compatible with *ptb.GetSecs()*
- **verbose** (*bool*) – Set to *True* to view the calculations along the way

getHandTriggerValues (*controller='Touch'*, *deadzone=False*)

Get the values of the hand triggers.

Parameters

- **controller** (*str*) – Name of the controller to get hand trigger values. Possible values for *controller* are ‘Touch’, ‘RTouch’, ‘LTouch’, ‘Object0’, ‘Object1’, ‘Object2’, and ‘Object3’; the only devices with hand triggers the SDK manages. For additional controllers, use PsychoPy’s built-in event or hardware support.
- **deadzone** (*bool*) – Apply the deadzone to hand trigger values. This pre-filters stick input to apply a dead-zone where 0.0 will be returned if the trigger returns a displacement within 0.2746.

Returns Left and right hand trigger values. Displacements are represented as *tuple* of two float representing the left and right displacement values, which range from 0.0 to 1.0. The returned values reflect the controller state since the last *updateInputState* or *flip* call.

Return type *tuple*

getIndexTriggerValues (*controller='Xbox', deadzone=False*)

Get the values of the index triggers.

Parameters

- **controller** (*str*) – Name of the controller to get index trigger values. Possible values for *controller* are ‘Xbox’, ‘Touch’, ‘RTouch’, ‘LTouch’, ‘Object0’, ‘Object1’, ‘Object2’, and ‘Object3’; the only devices with index triggers the SDK manages. For additional controllers, use PsychoPy’s built-in event or hardware support.
- **deadzone** (*bool*) – Apply the deadzone to index trigger values. This pre-filters stick input to apply a dead-zone where 0.0 will be returned if the trigger returns a displacement within 0.2746.

Returns Left and right index trigger values. Displacements are represented as *tuple* of two float representing the left and right displacement values, which range from 0.0 to 1.0. The returned values reflect the controller state since the last *updateInputState* or *flip* call.

Return type *tuple* of float

getMovieFrame (*buffer='mirror'*)

Capture the current HMD frame as an image.

Saves to stack for *saveMovieFrames()*. As of v1.81.00 this also returns the frame as a PIL image.

This can be done at any time (usually after a *flip()* command).

Frames are stored in memory until a *saveMovieFrames()* command is issued. You can issue *getMovieFrame()* as often as you like and then save them all in one go when finished.

For HMD frames, you should call *getMovieFrame* after calling *flip* to ensure that the mirror texture saved reflects what is presently being shown on the HMD. Note, that this function is somewhat slow and may impact performance. Only call this function when you’re not collecting experimental data.

Parameters **buffer** (*str, optional*) – Buffer to capture. For the HMD, only ‘mirror’ is available at this time.

Returns Buffer pixel contents as a PIL/Pillow image object.

Return type Image

getMsPerFrame (*nFrames=60, showVisual=False, msg="", msDelay=0.0*)

Assesses the monitor refresh rate (average, median, SD) under current conditions, over at least 60 frames.

Records time for each refresh (frame) for *n* frames (at least 60), while displaying an optional visual. The visual is just eye-candy to show that something is happening when assessing many frames. You can also give it text to display instead of a visual, e.g., *msg='(testing refresh rate...)'*; setting *msg* implies *showVisual == False*.

To simulate refresh rate under cpu load, you can specify a time to wait within the loop prior to doing the `flip()`. If $0 < \text{msDelay} < 100$, wait for that long in ms.

Returns timing stats (in ms) of:

- average time per frame, for all frames
- standard deviation of all frames
- median, as the average of 12 frame times around the median (~monitor refresh rate)

Author

- 2010 written by Jeremy Gray

`getPredictedDisplayTime()`

Get the predicted time the next frame will be displayed on the HMD. The returned time is referenced to the clock `LibOVR` is using.

Returns Absolute frame mid-point time for the given frame index in seconds.

Return type `float`

`getThumbstickValues(controller='Xbox', deadzone=False)`

Get controller thumbstick values.

Parameters

- **controller** (`str`) – Name of the controller to get thumbstick values. Possible values for `controller` are 'Xbox', 'Touch', 'RTouch', 'LTouch', 'Object0', 'Object1', 'Object2', and 'Object3'; the only devices with thumbsticks the SDK manages. For additional controllers, use PsychoPy's built-in event or hardware support.
- **deadzone** (`bool`) – Apply the deadzone to thumbstick values. This pre-filters stick input to apply a dead-zone where 0.0 will be returned if the sticks return a displacement within -0.2746 to 0.2746.

Returns Left and right, X and Y thumbstick values. Axis displacements are represented in each tuple by floats ranging from -1.0 (full left/down) to 1.0 (full right/up). The returned values reflect the controller state since the last `updateInputState` or `flip` call.

Return type `tuple`

`getTimeInSeconds()`

Absolute time in seconds. The returned time is referenced to the clock `LibOVR` is using.

Returns Time in seconds.

Return type `float`

`getTouches(touches, controller='Touch', testState='continuous')`

Get touch states from a controller.

Returns `True` if any names specified to `touches` reflect `testState` since the last `updateInputState` or `flip` call. If multiple button names are specified as a `list` or `tuple` to `touches`, multiple button states are tested, returning `True` if all the touches presently satisfy the `testState`. Note that not all controllers available support touches. If a touch is not supported or available, this function will always return `False`.

Special states can be used for basic gesture recognition, such as 'LThumbUp', 'RThumbUp', 'LIndex-Pointing', and 'RIndexPointing'.

Parameters

- **touches** (*list of str or str*) – Buttons to test. Valid *touches* names are ‘A’, ‘B’, ‘RThumb’, ‘RThumbRest’, ‘RThumbUp’, ‘RIndexPointing’, ‘LThumb’, ‘LThumbRest’, ‘LThumbUp’, ‘LIndexPointing’, ‘X’, and ‘Y’. Names can be passed as a *list* to test multiple button states.
- **controller** (*str*) – Controller name.
- **testState** (*str*) – State to test. Valid values are:
 - **continuous** - User is touching something on the controller.
 - **rising or pressed** - User began touching something since the last call to *updateInputState*.
 - **falling or released** - User stopped touching something since the last call to *updateInputState*.

Returns Touch state and timestamp in seconds the controller was polled.

Return type tuple of bool, float

Examples

Check if the ‘Enter’ button on the Oculus remote was released:

```
isPressed, tsec = hmd.getButtons(['Enter'], 'Remote', 'falling')
```

Check if the ‘A’ button was pressed on the touch controller:

```
isPressed, tsec = hmd.getButtons(['A'], 'Touch', 'pressed')
```

getTrackerInfo (*trackerIdx*)

Get tracker information.

Parameters **trackerIdx** (*int*) – Tracker index, ranging from 0 to *trackerCount*.

Returns Object containing tracker information.

Return type LibOVRTrackerInfo

Raises **IndexError** – Raised when *trackerIdx* out of range.

getTrackingState (*absTime=None, latencyMarker=True*)

Get the tracking state of the head and hands.

Calling this function retrieves the tracking state of the head (HMD) and hands at *absTime* from the *LibOVR* runtime. The returned object is a *LibOVRTrackingState* instance with poses, motion derivatives (i.e. linear and angular velocity/acceleration), and tracking status flags accessible through its attributes.

The pose states of the head and hands are available by accessing the *headPose* and *handPoses* attributes, respectively.

Parameters

- **absTime** (*float, optional*) – Absolute time the the tracking state refers to. If not specified, the predicted display time is used.
- **latencyMarker** (*bool, optional*) – Set a latency marker upon getting the tracking state. This is used for motion-to-photon calculations.

Returns Tracking state object. For more information about this type see:

Return type LibOVRTrackingState

See also:

`getPredictedDisplayTime()` Time at mid-frame for the current frame index.

Examples

Get the tracked head pose and use it to calculate render eye poses:

```
# get tracking state at predicted mid-frame time
absTime = getPredictedDisplayTime()
trackingState = hmd.getTrackingState(absTime)

# get the head pose from the tracking state
headPose = trackingState.headPose.thePose
hmd.calcEyePoses(headPose) # compute eye poses
```

Get linear/angular velocity and acceleration vectors of the right touch controller:

```
# right hand is the second value (index 1) at `handPoses`
rightHandState = trackingState.handPoses[1] # is `LibOVRPoseState`

# access `LibOVRPoseState` fields to get the data
linearVel = rightHandState.linearVelocity # m/s
angularVel = rightHandState.angularVelocity # rad/s
linearAcc = rightHandState.linearAcceleration # m/s^2
angularAcc = rightHandState.angularAcceleration # rad/s^2

# extract components like this if desired
vx, vy, vz = linearVel
ax, ay, az = angularVel
```

Above is useful for physics simulations, where one can compute the magnitude and direction of a force applied to a virtual object.

It's often the case that object tracking becomes unreliable for some reason, for instance, if it becomes occluded and is no longer visible to the sensors. In such cases, the reported pose state is invalid and may not be useful. You can check if the position and orientation of a tracked object is invalid using flags associated with the tracking state. This shows how to check if head position and orientation tracking was valid when sampled:

```
if trackingState.positionValid and trackingState.orientationValid:
    print('Tracking valid.')
```

It's up to the programmer to determine what to do in such cases. Note that tracking may still be valid even if

Get the calibrated origin used for tracking during the sample period of the tracking state:

```
calibratedOrigin = trackingState.calibratedOrigin
calibPos, calibOri = calibratedOrigin.posOri
```

Time integrate a tracking state. This extrapolates the pose over time given the present computed motion derivatives. The contrived example below shows how to implement head pose forward prediction:

```
# get current system time
absTime = getTimeInSeconds()

# get the elapsed time from `absTime` to predicted v-sync time,
# again this is an example, you would usually pass predicted time to
```

(continues on next page)

(continued from previous page)

```

# `getTrackingState` directly.
dt = getPredictedDisplayTime() - absTime

# get the tracking state for the current time, poses will lag where
# they are expected at predicted time by `dt` seconds
trackingState = hmd.getTrackingState(absTime)

# time integrate a pose by `dt`
headPoseState = trackingState.headPose
headPosePredicted = headPoseState.timeIntegrate(dt)

# calc eye poses with predicted head pose, this is a custom pose to
# head-locking should be enabled!
hmd.calcEyePoses(headPosePredicted)
    
```

The resulting head pose is usually very close to what `getTrackingState` would return if the predicted time was used. Simple forward prediction with time integration becomes increasingly unstable as the prediction interval increases. Under normal circumstances, let the runtime handle forward prediction by using the pose states returned at the predicted display time. If you plan on doing your own forward prediction, you need enable head-locking, clamp the prediction interval, and apply some sort of smoothing to keep the image as stable as possible.

property hasInputFocus

True if the application currently has input focus.

property hasMagYawCorrection

True if this HMD supports yaw drift correction.

property hasOrientationTracking

True if the HMD is capable of tracking orientation.

property hasPositionTracking

True if the HMD is capable of tracking position.

property headLocked

True if head locking is enabled.

property hid

USB human interface device (HID) identifiers (*int, int*).

hidePerfHud()

Hide the performance HUD.

property hmdMounted

True if the HMD is mounted on the user's head.

property hmdPresent

True if the HMD is present.

property isBoundaryVisible

True if the VR boundary is visible.

isPoseVisible (*pose*)

Check if a pose object is visible to the present eye. This method can be used to perform visibility culling to avoid executing draw commands for objects that fall outside the FOV for the current eye buffer.

If `boundingBox` has a valid bounding box object, this function will return *False* if all the box points fall completely to one side of the view frustum. If `boundingBox` is *None*, the point at `pos` is checked, returning *False* if it falls outside of the frustum. If the present buffer is not 'left' or 'right', this function will always return *False*.

Parameters `pose` (`LibOVRPose`) – Pose to test for visibility.

Returns `True` if pose’s bounding box or origin is outside of the view frustum.

Return type `bool`

property isVisible

`True` if the app has focus in the HMD and is visible to the viewer.

property lights

Scene lights.

This is specified as an array of `~psychopy.visual.LightSource` objects. If a single value is given, it will be converted to a `list` before setting. Set `useLights` to `True` before rendering to enable lighting/shading on subsequent objects. If `lights` is `None` or an empty `list`, no lights will be enabled if `useLights=True`, however, the scene ambient light set with `ambientLight` will be still be used.

Examples

Create a directional light source and add it to scene lights:

```
dirLight = gltools.LightSource((0., 1., 0.), lightType='directional')
win.lights = dirLight # `win.lights` will be a list when accessed!
```

Multiple lights can be specified by passing values as a list:

```
myLights = [gltools.LightSource((0., 5., 0.)),
            gltools.LightSource((-2., -2., 0.))]
win.lights = myLights
```

logOnFlip (`msg`, `level`, `obj=None`)

Send a log message that should be time-stamped at the next `flip()` command.

Parameters

- `msg` (`str`) – The message to be logged.
- `level` (`int`) – The level of importance for the message.
- `obj` (`object`, *optional*) – The python object that might be associated with this message if desired.

property manufacturer

Get the connected HMD’s manufacturer (`str`).

mouseVisible

Sets the visibility of the mouse cursor.

If Window was initialized with `allowGUI=False` then the mouse is initially set to invisible, otherwise it will initially be visible.

Usage:

```
win.mouseVisible = False
win.mouseVisible = True
```

multiFlip (`flips=1`, `clearBuffer=True`)

Flip multiple times while maintaining the display constant. Use this method for precise timing.

WARNING: This function should not be used. See the *Notes* section for details.

Parameters

- **flips** (*int, optional*) – The number of monitor frames to flip. Floats will be rounded to integers, and a warning will be emitted. `Window.multiFlip(flips=1)` is equivalent to `Window.flip()`. Defaults to *1*.
- **clearBuffer** (*bool, optional*) – Whether to clear the screen after the last flip. Defaults to *True*.

Notes

- This function can behave unpredictably, and the PsychoPy authors recommend against using it. See <https://github.com/psychopy/psychopy/issues/867> for more information.

Examples

Example of using `multiFlip`:

```
# Draws myStim1 to buffer
myStim1.draw()
# Show stimulus for 4 frames (90 ms at 60Hz)
myWin.multiFlip(clearBuffer=False, flips=6)
# Draw myStim2 "on top of" myStim1
# (because buffer was not cleared above)
myStim2.draw()
# Show this for 2 frames (30 ms at 60Hz)
myWin.multiFlip(flips=2)
# Show blank screen for 3 frames (buffer was cleared above)
myWin.multiFlip(flips=3)
```

multiplyProjectionMatrixGL()

Multiply the current projection modelMatrix obtained from the SDK using `glMultMatrixf`. The projection matrix used depends on the current eye buffer set by `setBuffer()`.

multiplyViewMatrixGL()

Multiply the local eye pose transformation modelMatrix obtained from the SDK using `glMultMatrixf`. The modelMatrix used depends on the current eye buffer set by `setBuffer()`.

Returns

Return type `None`

property nearClip

Distance to the near clipping plane in meters.

nextEditable()

Moves focus of the cursor to the next editable window

onResize (*width, height*)

A default resize event handler.

This default handler updates the GL viewport to cover the entire window and sets the `GL_PROJECTION` matrix to be orthogonal in window space. The bottom-left corner is (0, 0) and the top-right corner is the width and height of the *Window* in pixels.

Override this event handler with your own to create another projection, for example in perspective.

property overlayPresent

perfHudMode (*mode='Off'*)

Set the performance HUD mode.

Parameters `mode` (*str*) – HUD mode to use.

property `pixelsPerTanAngleAtCenter`

Horizontal and vertical pixels per tangent angle (=1) at the center of the display.

This can be used to compute pixels-per-degree for the display.

property `productName`

Get the HMD's product name (*str*).

property `projectionMatrix`

Get the projection matrix for the current eye buffer. Note that setting *projectionMatrix* manually will break visibility culling.

recenterTrackingOrigin ()

Recenter the tracking origin using the current head position.

recordFrameIntervals

Record time elapsed per frame.

Provides accurate measures of frame intervals to determine whether frames are being dropped. The intervals are the times between calls to `flip()`. Set to *True* only during the time-critical parts of the script. Set this to *False* while the screen is not being updated, i.e., during any slow, non-frame-time-critical sections of your code, including `inter-trial-intervals`, `event.waitkeys()`, `core.wait()`, or `image.setImage()`.

Examples

Enable frame interval recording, successive frame intervals will be stored:

```
win.recordFrameIntervals = True
```

Frame intervals can be saved by calling the `saveFrameIntervals` method:

```
win.saveFrameIntervals()
```

removeEditable (*editable*)

resetEyeTransform (*clearDepth=True*)

Restore the default projection and view settings to PsychoPy defaults. Call this prior to drawing 2D stimuli objects (i.e. `GratingStim`, `ImageStim`, `Rect`, etc.) if any eye transformations were applied for the stimuli to be drawn correctly.

Parameters `clearDepth` (*bool*) – Clear the depth buffer upon reset. This ensures successive draw commands are not affected by previous data written to the depth buffer. Default is *True*.

Notes

- Calling `flip()` automatically resets the view and projection to defaults. So you don't need to call this unless you are mixing 3D and 2D stimuli.

Examples

Going between 3D and 2D stimuli:

```
# 2D stimuli can be drawn before setting a perspective projection
win.setPerspectiveView()
# draw 3D stimuli here ...
win.resetEyeTransform()
# 2D stimuli can be drawn here again ...
win.flip()
```

resetViewport ()

Reset the viewport to cover the whole framebuffer.

Set the viewport to match the dimensions of the back buffer or framebuffer (if *useFBO=True*). The scissor rectangle is also set to match the dimensions of the viewport.

property rgb

saveFrameIntervals (*fileName=None, clear=True*)

Save recorded screen frame intervals to disk, as comma-separated values.

Parameters

- **fileName** (*None* or *str*) – *None* or the filename (including path if necessary) in which to store the data. If *None* then ‘lastFrameIntervals.log’ will be used.
- **clear** (*bool*) – Clear buffer frames intervals were stored after saving. Default is *True*.

saveMovieFrames (*fileName, codec='libx264', fps=30, clearFrames=True*)

Writes any captured frames to disk.

Will write any format that is understood by PIL (tif, jpg, png, ...)

Parameters

- **filename** (*str*) – Name of file, including path. The extension at the end of the file determines the type of file(s) created. If an image type (e.g. .png) is given, then multiple static frames are created. If it is .gif then an animated GIF image is created (although you will get higher quality GIF by saving PNG files and then combining them in dedicated image manipulation software, such as GIMP). On Windows and Linux .mpeg files can be created if *pymedia* is installed. On macOS .mov files can be created if the *pyobjc-frameworks-QtKit* is installed. Unfortunately the libs used for movie generation can be flaky and poor quality. As for animated GIFs, better results can be achieved by saving as individual .png frames and then combining them into a movie using software like *ffmpeg*.
- **codec** (*str, optional*) – The codec to be used by **moviepy** for mp4/mpg/mov files. If *None* then the default will depend on file extension. Can be one of *libx264*, *mpeg4* for mp4/mov files. Can be *rawvideo*, *png* for avi files (not recommended). Can be *libvorbis* for ogv files. Default is *libx264*.
- **fps** (*int, optional*) – The frame rate to be used throughout the movie. **Only for quicktime (.mov) movies.** Default is *30*.
- **clearFrames** (*bool, optional*) – Set this to *False* if you want the frames to be kept for additional calls to *saveMovieFrames*. Default is *True*.

Examples

Writes a series of static frames as frame001.tif, frame002.tif etc.:

```
myWin.saveMovieFrames('frame.tif')
```

As of PsychoPy 1.84.1 the following are written with moviepy:

```
myWin.saveMovieFrames('stimuli.mp4') # codec = 'libx264' or 'mpeg4'
myWin.saveMovieFrames('stimuli.mov')
myWin.saveMovieFrames('stimuli.gif')
```

property scissor

Scissor rectangle (x, y, w, h) for the current draw buffer.

Values *x* and *y* define the origin, and *w* and *h* the size of the rectangle in pixels. The scissor operation is only active if *scissorTest=True*.

Usually, the scissor and viewport are set to the same rectangle to prevent drawing operations from *spilling* into other regions of the screen. For instance, calling *clearBuffer* will only clear within the scissor rectangle.

Setting the scissor rectangle but not the viewport will restrict drawing within the defined region (like a rectangular aperture), not changing the positions of stimuli.

property scissorTest

True if scissor testing is enabled.

property screenshot

property sensorSampleTime

Sensor sample time (*float*). This value corresponds to the time the head (HMD) position was sampled, which is required for computing motion-to-photon latency. This does not need to be specified if *getTrackingState* was called with *latencyMarker=True*.

property serialNumber

Get the connected HMD's unique serial number (*str*).

Use this to identify a particular unit if you own many.

setBlendMode (blendMode, log=None)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBuffer (buffer, clear=True)

Set the active draw buffer.

Warning: The window.Window.size property will return the buffer's dimensions in pixels instead of the window's when *setBuffer* is set to 'left' or 'right'.

Parameters

- **buffer** (*str*) – View buffer to divert successive drawing operations to, can be either 'left' or 'right'.
- **clear** (*boolean*) – Clear the color, stencil and depth buffer.

setColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Usually you can use `stim.attribute = value` syntax instead, but use this method if you want to set color and colorSpace simultaneously.

See `color` for documentation on colors.

setDefaultView (*clearDepth=True*)

Return to default projection. Call this before drawing PsychoPy's 2D stimuli after a stereo projection change.

Note: This only has an effect if using Rift in legacy immediate mode OpenGL.

Parameters `clearDepth` (*bool*) – Clear the depth buffer prior after configuring the view parameters.

setGamma (*gamma*, *log=None*)

Usually you can use `'stim.attribute = value'` syntax instead, but use this method if you need to suppress the log message.

setMouseType (*name='arrow'*)

Change the appearance of the cursor for this window. Cursor types provide contextual hints about how to interact with on-screen objects.

The graphics used 'standard cursors' provided by the operating system. They may vary in appearance and hot spot location across platforms. The following names are valid on most platforms:

- `arrow`: Default pointer.
- `ibeam`: Indicates text can be edited.
- `crosshair`: Crosshair with hot-spot at center.
- `hand`: A pointing hand.
- `hresize`: Double arrows pointing horizontally.
- `vresize`: Double arrows pointing vertically.

Parameters `name` (*str*) – Type of standard cursor to use (see above). Default is `arrow`.

Notes

- On Windows the `crosshair` option is negated with the background color. It will not be visible when placed over 50% grey fields.

setMouseVisible (*visibility*, *log=None*)

Usually you can use `'stim.attribute = value'` syntax instead, but use this method if you need to suppress the log message.

setOffAxisView (*applyTransform=True*, *clearDepth=True*)

Set an off-axis projection.

Create an off-axis projection for subsequent rendering calls. Sets the `viewMatrix` and `projectionMatrix` accordingly so the scene origin is on the screen plane. If `eyeOffset` is correct and the view distance and screen size is defined in the monitor configuration, the resulting view will approximate *ortho-stereo* viewing.

The convergence plane can be adjusted by setting `convergeOffset`. By default, the convergence plane is set to the screen plane. Any points on the screen plane will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards.
- **clearDepth** (*bool*, *optional*) – Clear the depth buffer.

setPerspectiveView (*applyTransform=True*, *clearDepth=True*)

Set the projection and view matrix to render with perspective.

Matrices are computed using values specified in the monitor configuration with the scene origin on the screen plane. Calculations assume units are in meters. If *eyeOffset* $\neq 0$, the view will be transformed laterally, however the frustum shape will remain the same.

Note that the values of `projectionMatrix` and `viewMatrix` will be replaced when calling this function.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards if *False*.
- **clearDepth** (*bool*, *optional*) – Clear the depth buffer.

setRGB (*newRGB*)

Deprecated: As of v1.61.00 please use `setColor()` instead

setRecordFrameIntervals (*value=True*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setRiftView (*clearDepth=True*)

Set head-mounted display view. Gets the projection and view matrices from the HMD and applies them.

Note: This only has an effect if using Rift in legacy immediate mode OpenGL.

Parameters **clearDepth** (*bool*) – Clear the depth buffer prior after configuring the view parameters.

setScale (*units*, *font='dummyFont'*, *prevScale=1.0*, *1.0*)

DEPRECATED: this method used to be used to switch between units for stimulus drawing but this is now handled by the stimuli themselves and the window should always be left in units of ‘pix’

setSize (*value*, *log=True*)

setStereoDebugHudOption (*option*, *value*)

Configure stereo debug HUD guides.

Parameters

- **option** (*str*) – Option to set. Valid options are *InfoEnable*, *Size*, *Position*, *YawPitchRoll*, and *Color*.
- **value** (*array_like* or *bool*) – Value to set for a given *option*. Appropriate types for each option are:
 - *InfoEnable* - bool, *True* to show, *False* to hide.
 - *Size* - array_like, [w, h] in meters.
 - *Position* - array_like, [x, y, z] in meters.
 - *YawPitchRoll* - array_like, [pitch, yaw, roll] in degrees.
 - *Color* - array_like, [r, g, b] as floats ranging 0.0 to 1.0.

Returns `True` if the option was successfully set.

Return type bool

Examples

Configuring a stereo debug HUD guide:

```
# show a quad with a crosshair
hmd.stereoDebugHudMode('QuadWithCrosshair')
# enable displaying guide information
hmd.setStereoDebugHudOption('InfoEnable', True)
# set the position of the guide quad in the scene
hmd.setStereoDebugHudOption('Position', [0.0, 1.7, -2.0])
```

setToeInView (*applyTransform=True, clearDepth=True*)

Set toe-in projection.

Create a toe-in projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. The value of *convergeOffset* will define the convergence point of the view, which is offset perpendicular to the center of the screen plane. Points falling on a vertical line at the convergence point will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

Notes

- This projection mode is only ‘correct’ if the viewer’s eyes are converged at the convergence point. Due to perspective, this projection introduces vertical disparities which increase in magnitude with eccentricity. Use *setOffAxisView* if you want to display something the viewer can look around the screen comfortably.

setUnits (*value, log=True*)

setViewPos (*value, log=True*)

property shouldQuit

True if the user requested the application should quit through the headset’s interface.

property shouldRecenter

True if the user requested the origin be re-centered through the headset’s interface.

property size

Size property to get the dimensions of the view buffer instead of the window. If there are no view buffers, always return the dims of the window.

specifyTrackingOrigin (*pose*)

Specify a tracking origin. If *trackingOriginType='floor'*, this function sets the origin of the scene in the ground plane. If *trackingOriginType='eye'*, the scene origin is set to the known eye height.

Parameters pose (*LibOVRPose*) – Tracking origin pose.

specifyTrackingOriginPosOri (*pos=0.0, 0.0, 0.0, ori=0.0, 0.0, 0.0, 1.0*)

Specify a tracking origin using a pose and orientation. This is the same as *specifyTrackingOrigin*, but accepts a position vector [x, y, z] and orientation quaternion [x, y, z, w].

Parameters

- **pos** (*tuple or list of float, or ndarray*) – Position coordinate of origin (x, y, z).
- **ori** (*tuple or list of float, or ndarray*) – Quaternion specifying orientation (x, y, z, w).

startHaptics (*controller, frequency='low', amplitude=1.0*)

Start haptic feedback (vibration).

Vibration is constant at fixed frequency and amplitude. Vibration lasts 2.5 seconds, so this function needs to be called more often than that for sustained vibration. Only controllers which support vibration can be used here.

There are only two frequencies permitted 'high' and 'low', however, amplitude can vary from 0.0 to 1.0. Specifying 'frequency'='off' stops vibration if in progress.

Parameters

- **controller** (*str*) – Name of the controller to vibrate.
- **frequency** (*str*) – Vibration frequency. Valid values are: 'off', 'low', or 'high'.
- **amplitude** (*float*) – Vibration amplitude in the range of [0.0 and 1.0]. Values outside this range are clamped.

property stencilTest

True if stencil testing is enabled.

stereoDebugHudMode (*mode*)

Set the debug stereo HUD mode.

This makes the compositor add stereoscopic reference guides to the scene. You can configure the HUD can be configured using other methods.

Parameters mode (*str*) – Stereo debug mode to use. Valid options are *Off*, *Quad*, *QuadWithCrosshair*, and *CrosshairAtInfinity*.

Examples

Enable a stereo debugging guide:

```
hmd.stereoDebugHudMode('CrosshairAtInfinity')
```

Hide the debugging guide. Should be called before exiting the application since it's persistent until the Oculus service is restarted:

```
hmd.stereoDebugHudMode('Off')
```

stopHaptics (*controller*)

Stop haptic feedback.

Convenience function to stop controller vibration initiated by the last `vibrateController` call. This is the same as calling `vibrateController(controller, frequency='off')`.

Parameters controller (*str*) – Name of the controller to stop vibrating.

submitControllerVibration (*controller, hapticsBuffer*)

Submit a haptics buffer to begin controller vibration.

Parameters

- **controller** (*str*) – Name of controller to vibrate.
- **hapticsBuffer** (*LibOVRHapticsBuffer*) – Haptics buffer to playback.

Notes

Methods *startHaptics* and *stopHaptics* cannot be used interchangeably with this function.

tanAngleToNDC (*horzTan, vertTan*)

Convert tan angles to the normalized device coordinates for the current buffer.

Parameters

- **horzTan** (*float*) – Horizontal tan angle.
- **vertTan** (*float*) – Vertical tan angle.

Returns Normalized device coordinates X, Y. Coordinates range between -1.0 and 1.0. Returns *None* if an invalid buffer is selected.

Return type tuple of float

testBoundary (*deviceType, bounadryType='PlayArea'*)

Test if tracked devices are colliding with the play area boundary.

This returns an object containing test result data.

Parameters

- **deviceType** (*str, list or tuple*) – The device to check for boundary collision. If a list of names is provided, they will be combined and all tested.
- **boundaryType** (*str*) – Boundary type to test.

timeOnFlip (*obj, attrib*)

Retrieves the time on the next flip and assigns it to the *attrib* for this *obj*.

Parameters

- **obj** (*dict or object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of *obj* to assign the flip time to.

Examples

Assign time on flip to the `tStartRefresh` key of `myTimingDict`:

```
win.getTimeOnFlip(myTimingDict, 'tStartRefresh')
```

property trackerCount

Number of attached trackers.

property trackingOriginType

Current tracking origin type (*str*).

Valid tracking origin types are 'floor' and 'eye'.

units

None, 'height' (of the window), 'norm', 'deg', 'cm', 'pix' Defines the default units of stimuli initialized in the window. I.e. if you change units, already initialized stimuli won't change their units.

Can be overridden by each stimulus, if units is specified on initialization.

See *Units for the window and stimuli* for explanation of options.

update ()

Deprecated: use `Window.flip()` instead

updateInputState (*controllers=None*)

Update all connected controller states. This updates controller input states for an input device managed by *LibOVR*.

The polling time for each device is accessible through the *controllerPollTimes* attribute. This attribute returns a dictionary where the polling time from the last *updateInputState* call for a given controller can be retrieved by using the name as a key.

Parameters **controllers** (*tuple or list, optional*) – List of controllers to poll. If *None*, all available controllers will be polled.

Examples

Poll the state of specific controllers by name:

```
controllers = ['XBox', 'Touch']
updateInputState(controllers)
```

updateLights (*index=None*)

Explicitly update scene lights if they were modified.

This is required if modifications to objects referenced in *lights* have been changed since assignment. If you removed or added items of *lights* you must refresh all of them.

Parameters **index** (*int, optional*) – Index of light source in *lights* to update. If *None*, all lights will be refreshed.

Examples

Call *updateLights* if you modified lights directly like this:

```
win.lights[1].diffuseColor = [1., 0., 0.]
win.updateLights(1)
```

property useLights

Enable scene lighting.

Lights will be enabled if using legacy OpenGL lighting. Stimuli using shaders for lighting should check if *useLights* is *True* since this will have no effect on them, and disable or use a no lighting shader instead. Lights will be transformed to the current view matrix upon setting to *True*.

Lights are transformed by the present *GL_MODELVIEW* matrix. Setting *useLights* will result in their positions being transformed by it. If you want lights to appear at the specified positions in world space, make sure the current matrix defines the view/eye transformation when setting *useLights=True*.

This flag is reset to *False* at the beginning of each frame. Should be *False* if rendering 2D stimuli or else the colors will be incorrect.

property userHeight

Get user height in meters (*float*).

property viewMatrix

The view matrix for the current eye buffer. Only valid after a *calcEyePoses()* call. Note that setting *viewMatrix* manually will break visibility culling.

viewPos

The origin of the window onto which stimulus-objects are drawn.

The value should be given in the units defined for the window. NB: Never change a single component (x or y) of the origin, instead replace the viewPos-attribute in one shot, e.g.:

```
win.viewPos = [new_xval, new_yval] # This is the way to do it
win.viewPos[0] = new_xval # DO NOT DO THIS! Errors will result.
```

property viewport

Viewport rectangle (x, y, w, h) for the current draw buffer.

Values *x* and *y* define the origin, and *w* and *h* the size of the rectangle in pixels.

This is typically set to cover the whole buffer, however it can be changed for applications like multi-view rendering. Stimuli will draw according to the new shape of the viewport, for instance and stimulus with position (0, 0) will be drawn at the center of the viewport, not the window.

Examples

Constrain drawing to the left and right halves of the screen, where stimuli will be drawn centered on the new rectangle. Note that you need to set both the *viewport* and the *scissor* rectangle:

```
x, y, w, h = win.frameBufferSize # size of the framebuffer
win.viewport = win.scissor = [x, y, w / 2.0, h]
# draw left stimuli ...

win.viewport = win.scissor = [x + (w / 2.0), y, w / 2.0, h]
# draw right stimuli ...

# restore drawing to the whole screen
win.viewport = win.scissor = [x, y, w, h]
```

waitBlanking

After a call to `flip()` should we wait for the blank before the script continues.

property windowedSize

Size of the window to use when not fullscreen (w, h).

9.3.27 RigidBodyPose

Attributes

<i>RigidBodyPose</i> ([pos, ori])	Class for representing rigid body poses.
-----------------------------------	--

Details

class `psychopy.visual.RigidBodyPose` (*pos=0.0, 0.0, 0.0, ori=0.0, 0.0, 0.0, 1.0*)

Class for representing rigid body poses.

This class is an abstract representation of a rigid body pose, where the position of the body in a scene is represented by a vector/coordinate and the orientation with a quaternion. Pose can be manipulated and interacted with using class methods and attributes. Rigid body poses assume a right-handed coordinate system (-Z is forward and +Y is up).

Poses can be converted to 4x4 transformation matrices with `getModelMatrix`. One can use these matrices when rendering to transform the vertices of a model associated with the pose by passing them to OpenGL. Matrices are cached internally to avoid recomputing them if *pos* and *ori* attributes have not been updated.

Operators `*` and `~` can be used on *RigidBodyPose* objects to combine and invert poses. For instance, you can multiply (`*`) poses to get a new pose which is the combination of both orientations and translations by:

```
newPose = rb1 * rb2
```

Likewise, a pose can be inverted by using the `~` operator:

```
invPose = ~rb
```

Multiplying a pose by its inverse will result in an identity pose with no translation and default orientation where *pos*=[0, 0, 0] and *ori*=[0, 0, 0, 1]:

```
identityPose = ~rb * rb
```

Warning: This class is experimental and may result in undefined behavior.

Parameters

- **pos** (*array_like*) – Position vector $[x, y, z]$ for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion $[x, y, z, w]$ where x, y, z are imaginary and w is real.

alignTo (*alignTo*)

Align this pose to another point or pose.

This sets the orientation of this pose to one which orients the forward axis towards *alignTo*.

Parameters **alignTo** (*array_like* or *LibOVRPose*) – Position vector $[x, y, z]$ or pose to align to.

property at

Vector defining the forward direction (-Z) of this pose.

property bounds

Bounding box associated with this pose.

copy ()

Get a new *RigidBodyPose* object which copies the position and orientation of this one. Copies are independent and do not reference each others data.

Returns Copy of this pose.

Return type *RigidBodyPose*

distanceTo (*v*)

Get the distance to a pose or point in scene units.

Parameters *v* (*RigidBodyPose* or *array_like*) – Pose or point [x, y, z] to compute distance to.

Returns Distance to *v* from this pose’s origin.

Return type *float*

getModelMatrix (*inverse=False*, *out=None*)

Get the present rigid body transformation as a 4x4 matrix.

Matrices are computed only if the *pos* and *ori* attributes have been updated since the last call to *getModelMatrix*. The returned matrix is an *ndarray* and row-major.

Parameters

- **inverse** (*bool*, *optional*) – Return the inverse of the model matrix.
- **out** (*ndarray* or *None*) – Optional 4x4 array to write values to. Values written are computed using 32-bit float precision regardless of the data type of *out*.

Returns 4x4 transformation matrix.

Return type *ndarray*

Examples

Using a rigid body pose to transform something in OpenGL:

```
rb = RigidBodyPose((0, 0, -2)) # 2 meters away from origin

# Use `array2pointer` from `psychopy.tools.arraytools` to convert
# array to something OpenGL accepts.
mv = array2pointer(rb.modelMatrix)

# use the matrix to transform the scene
glMatrixMode(GL_MODELVIEW)
glPushMatrix()
glLoadIdentity()
glMultTransposeMatrixf(mv)

# draw the thing here ...

glPopMatrix()
```

getNormalMatrix (*out=None*)

Get the present normal matrix.

Parameters *out* (*ndarray* or *None*) – Optional 4x4 array to write values to. Values written are computed using 32-bit float precision regardless of the data type of *out*.

Returns 4x4 normal transformation matrix.

Return type *ndarray*

getOriAxisAngle (*degrees=True*)

Get the axis and angle of rotation for the rigid body. Converts the orientation defined by the *ori* quaternion to and axis-angle representation.

Parameters `degrees` (*bool, optional*) – Specify `True` if *angle* is in degrees, or else it will be treated as radians. Default is `True`.

Returns Axis [rx, ry, rz] and angle.

Return type `tuple`

getViewMatrix (*inverse=False*)

Convert this pose into a view matrix.

Creates a view matrix which transforms points into eye space using the current pose as the eye position in the scene. Furthermore, you can use view matrices for rendering shadows if light positions are defined as *RigidBodyPose* objects.

Parameters `inverse` (*bool*) – Return the inverse of the view matrix. Default is *False*.

Returns 4x4 transformation matrix.

Return type `ndarray`

getYawPitchRoll (*degrees=True*)

Get the yaw, pitch and roll angles for this pose relative to the -Z world axis.

Parameters `degrees` (*bool, optional*) – Specify `True` if *angle* is in degrees, or else it will be treated as radians. Default is `True`.

interp (*end, s*)

Interpolate between poses.

Linear interpolation is used on position (Lerp) while the orientation has spherical linear interpolation (Slerp) applied taking the shortest arc on the hypersphere.

Parameters

- `end` (*LibOVRPose*) – End pose.
- `s` (*float*) – Interpolation factor between interval 0.0 and 1.0.

Returns Rigid body pose whose position and orientation is at *s* between this pose and *end*.

Return type *RigidBodyPose*

property inverseModelMatrix

Inverse of the pose as a 4x4 model matrix (read-only).

invert ()

Invert this pose.

inverted ()

Get a pose which is the inverse of this one.

Returns This pose inverted.

Return type *RigidBodyPose*

isEqual (*other*)

Check if poses have similar orientation and position.

Parameters `other` (*RigidBodyPose*) – Other pose to compare.

Returns Returns *True* if poses are effectively equal.

Return type `bool`

property modelMatrix

Pose as a 4x4 model matrix (read-only).

property normalMatrix

The normal transformation matrix.

property ori

Orientation quaternion (X, Y, Z, W).

property pos

Position vector (X, Y, Z).

property posOri

The position (x, y, z) and orientation (x, y, z, w).

setIdentity()

Clear rigid body transformations.

setOriAxisAngle (*axis, angle, degrees=True*)

Set the orientation of the rigid body using an *axis* and *angle*. This sets the quaternion at *ori*.

Parameters

- **axis** (*array_like*) – Axis of rotation [rx, ry, rz].
- **angle** (*float*) – Angle of rotation.
- **degrees** (*bool, optional*) – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

transform (*v, out=None*)

Transform a vector using this pose.

Parameters

- **v** (*array_like*) – Vector to transform [x, y, z].
- **out** (*ndarray or None, optional*) – Optional array to write values to. Must have the same shape as *v*.

Returns Transformed points.

Return type ndarray

transformNormal (*n*)

Rotate a normal vector with respect to this pose.

Rotates a normal vector *n* using the orientation quaternion at *ori*.

Parameters **n** (*array_like*) – Normal to rotate (1-D with length 3).

Returns Rotated normal *n*.

Return type ndarray

property up

Vector defining the up direction (+Y) of this pose.

9.3.28 SceneSkybox

Attributes

<code>SceneSkybox(win[, tex, ori, axis])</code>	Class to render scene skyboxes.
---	---------------------------------

Details

class `psychopy.visual.SceneSkybox` (*win, tex=(), ori=0.0, axis=0, 1, 0*)
 Class to render scene skyboxes.

A skybox provides background imagery to serve as a visual reference for the scene. Background images are projected onto faces of a cube centered about the viewpoint regardless of any viewpoint translations, giving the illusion that the background is very far away. Usually, only one skybox can be rendered per buffer each frame. Render targets must have a depth buffer associated with them.

Background images are specified as a set of image paths passed to *faceTextures*:

```
sky = SceneSkybox(
    win, ('rt.jpg', 'lf.jpg', 'up.jpg', 'dn.jpg', 'bk.jpg', 'ft.jpg'))
```

The skybox is rendered by calling *draw()* after drawing all other 3D stimuli.

Skyboxes are not affected by lighting, however, their colors can be modulated by setting the window's *sceneAmbient* value. Skyboxes should be drawn after all other 3D stimuli, but before any successive call that clears the depth buffer (eg. *setPerspectiveView*, *resetEyeTransform*, etc.)

Parameters

- **win** (*~psychopy.visual.Window*) – Window this skybox is associated with.
- **tex** (*list or tuple or TexCubeMap*) – List of files paths to images to use for each face. Images are assigned to faces depending on their index within the list ([+X, -X, +Y, -Y, +Z, -Z] or [right, left, top, bottom, back, front]). If *None* is specified, the cube map may be specified later by setting the *cubemap* attribute. Alternatively, you can specify a *TexCubeMap* object to set the cube map directly.
- **ori** (*float*) – Rotation of the skybox about *axis* in degrees.
- **axis** (*array_like*) – Axis [ax, ay, az] to rotate about, default is (0, 1, 0).

draw (*win=None*)

Draw the skybox.

This should be called last after drawing other 3D stimuli for performance reasons.

Parameters **win** (*~psychopy.visual.Window*, optional) – Window to draw the skybox to. If *None*, the window set when initializing this object will be used. The window must share a context with the window which this objects was initialized with.

property `skyCubeMap`

Cubemap for the sky.

9.3.29 EnvelopeGrating

Attributes

Details

class psychopy.visual.**EnvelopeGrating** (*args, **kwargs)

Second-order envelope stimuli with 3 textures; a carrier, an envelope and a mask

Examples:

env1 = EnvelopeGrating(win,ori=0, carrier='sin', envelope='sin', mask = 'gauss', sf=24, envsf=4, size=1, contrast=0.5, moddepth=1.0, envori=0, pos=[-.5,.5],interpolate=0) # gives a circular patch of high frequency carrier with a # low frequency envelope

env2 = EnvelopeGrating(win,ori=0, carrier=noise, envelope='sin', mask = None, sf=1, envsf=4, size=1, contrast=0.5, moddepth=0.8, envori=0, pos=[-.5,-.5],interpolate=0) # If noise is some numpy array containing random values gives a # patch of noise with a low frequency sinewave envelope

env4 = EnvelopeGrating(win,ori=90, carrier='sin', envelope='sin', mask = 'gauss', sf=24, envsf=4, size=1, contrast=0.5, moddepth=1.0, envori=0, pos=[.5,.5], beat=True, interpolate=0) # Setting beat will create a second order beat stimulus which # critically contains no net energy at the carrier frequency # even though it appears to be present. In this case carrier # and envelope are at 90 degree to each other

With an EnvelopeStim the carrier and envelope can have different spatial frequencies, phases and orientations. Its position can be shifted as a whole.

contrast controls the contrast of the carrier and moddepth the modulation depth of the envelope. contrast and moddepth must work together, for moddepth=1 the max carrier contrast is 0.5 otherwise the displayable range will be exceeded. If moddepth < 1 higher contrasts can be accommodated.

Opacity controls the transparency of the whole stimulus.

Because orientation is implemented very differently for the carrier and envelope using this function without a broadly circular mask may produce unexpected results

Using EnvelopeStim with images from disk (jpg, tif, png, ...)

Ideally texture images to be rendered should be square with 'power-of-2' dimensions e.g. 16x16, 128x128. Any image that is not will be upscaled (with linear interpolation) to the nearest such texture by PsychoPy. The size of the stimulus should be specified in the normal way using the appropriate units (deg, pix, cm, ...). Be sure to get the aspect ratio the same as the image (if you don't want it stretched!).

9.3.30 psychopy.visual.ShapeStim

ShapeStim is the base class for drawing lines and polygons.

Overview

<i>ShapeStim</i> (win[, units, colorSpace, ...])	A class for arbitrary shapes defined as lists of vertices (x,y).
<i>ShapeStim</i> .units	
<i>ShapeStim</i> .lineWidth	Width of the line in pixels .
<i>ShapeStim</i> .lineColor	Alternative way of setting <i>borderColor</i> .
<i>ShapeStim</i> .fillColor	Set the fill color for the shape.

continues on next page

Table 9.22 – continued from previous page

<i>ShapeStim.colorSpace</i>	The name of the color space currently being used
<i>ShapeStim.vertices</i>	A list of lists or a numpy array (Nx2) specifying xy positions of each vertex, relative to the center of the field.
<i>ShapeStim.closeShape</i>	Should the last vertex be automatically connected to the first?
<i>ShapeStim.pos</i>	The position of the center of the stimulus in the stimulus <i>units</i>
<i>ShapeStim.size</i>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<i>ShapeStim.ori</i>	The orientation of the stimulus (in degrees).
<i>ShapeStim.opacity</i>	Determines how visible the stimulus is relative to background.
<i>ShapeStim.contrast</i>	A value that is simply multiplied by the color.
<i>ShapeStim.depth</i>	DEPRECATED, depth is now controlled simply by drawing order.
<i>ShapeStim.interpolate</i>	If <i>True</i> the edge of the line will be anti-aliased.
<i>ShapeStim.lineRGB</i>	Legacy property for setting the border color of a stimulus in RGB, instead use <i>obj._borderColor.rgb</i>
<i>ShapeStim.fillRGB</i>	Legacy property for setting the fill color of a stimulus in RGB, instead use <i>obj._fillColor.rgb</i>
<i>ShapeStim.name</i>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<i>ShapeStim.autoLog</i>	Whether every change in this stimulus should be auto logged.
<i>ShapeStim.autoDraw</i>	Determines whether the stimulus should be automatically drawn on every frame flip.
<i>ShapeStim.color</i>	Set the color of the shape.
<i>ShapeStim.lineColorSpace</i>	Deprecated, please use <i>colorSpace</i> to set color space for the entire object
<i>ShapeStim.fillColorSpace</i>	Deprecated, please use <i>colorSpace</i> to set color space for the entire object.

Details

class `psychopy.visual.shape.ShapeStim` (*win*, *units=""*, *colorSpace='rgb'*, *fillColor=False*, *lineColor=False*, *lineWidth=1.5*, *vertices=- 0.5, 0, 0, 0.5, 0.5, 0*, *windingRule=None*, *closeShape=True*, *pos=0, 0*, *size=1*, *anchor=None*, *ori=0.0*, *opacity=1.0*, *contrast=1.0*, *depth=0*, *interpolate=True*, *name=None*, *autoLog=None*, *autoDraw=False*, *color=False*, *lineRGB=False*, *fillRGB=False*, *fillColorSpace=None*, *lineColorSpace=None*)

A class for arbitrary shapes defined as lists of vertices (x,y).

Shapes can be lines, polygons (concave, convex, self-crossing), or have holes or multiple regions.

vertices is typically a list of points (x,y). By default, these are assumed to define a closed figure (polygon); set *closeShape=False* for a line. *closeShape* cannot be changed dynamically, but individual vertices can be changed on a frame-by-frame basis. The stimulus as a whole can be rotated, translated, or scaled dynamically (using *.ori*, *.pos*, *.size*).

Vertices can be a string, giving the name of a known set of vertices, although “cross” is the only named shape available at present.

Advanced shapes: *vertices* can also be a list of loops, where each loop is a list of points (x,y), e.g., to define a shape with a hole. Borders and *contains()* are not supported for multi-loop stimuli.

windingRule is an advanced feature to allow control over the GLU tessellator winding rule (default: `GLU_TESS_WINDING_ODD`). This is relevant only for self-crossing or multi-loop shapes. Cannot be set dynamically.

See Coder demo > stimuli > shapes.py

Changed Nov 2015: v1.84.00. Now allows filling of complex shapes. This is almost completely backwards compatible (see changelog). The old version is accessible as *psychopy.visual.BaseShapeStim*.

Parameters

- **win** (*Window*) – Window this shape is being drawn to. The stimulus instance will allocate its required resources using that Windows context. In many cases, a stimulus instance cannot be drawn on different windows unless those windows share the same OpenGL context, which permits resources to be shared between them.
- **units** (*str*) – Units to use when drawing. This will affect how parameters and attributes *pos*, *size* and *radius* are interpreted.
- **colorSpace** (*str*) – Sets the colorspace, changing how values passed to *lineColor* and *fillColor* are interpreted.
- **lineWidth** (*float*) – Width of the shape outline.
- **lineColor** (*array_like*, *str*, *Color* or *None*) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **fillColor** (*array_like*, *str*, *Color* or *None*) – Color of the shape outline and fill. If *None*, a fully transparent color is used which makes the fill or outline invisible.
- **vertices** (*array_like*) – Nx2 array of points (eg., *[[-0.5, 0], [0, 0.5], [0.5, 0]]*).
- **windingRule** (*GLenum* or *None*) – Winding rule to use for tessellation, default is `GLU_TESS_WINDING_ODD` if *None* is specified.
- **closeShape** (*bool*) – Close the shape's outline. If *True* the first and last vertex will be joined by an edge. Must be *True* to use tessellation. Default is *True*.
- **pos** (*array_like*) – Initial position (*x*, *y*) of the shape on-screen relative to the origin located at the center of the window or buffer in *units*. This can be updated after initialization by setting the *pos* property. The default value is *(0.0, 0.0)* which results in no translation.
- **size** (*array_like*, *float*, *int* or *None*) – Width and height of the shape as *(w, h)* or *[w, h]*. If a single value is provided, the width and height will be set to the same specified value. If *None* is specified, the *size* will be set with values passed to *width* and *height*.
- **ori** (*float*) – Initial orientation of the shape in degrees about its origin. Positive values will rotate the shape clockwise, while negative values will rotate counterclockwise. The default value for *ori* is 0.0 degrees.
- **opacity** (*float*) – Opacity of the shape. A value of 1.0 indicates fully opaque and 0.0 is fully transparent (therefore invisible). Values between 1.0 and 0.0 will result in colors being blended with objects in the background. This value affects the fill (*fillColor*) and outline (*lineColor*) colors of the shape.
- **contrast** (*float*) – Contrast level of the shape (0.0 to 1.0). This value is used to modulate the contrast of colors passed to *lineColor* and *fillColor*.
- **depth** (*int*) – Depth layer to draw the shape when *autoDraw* is enabled. *DEPRECATED*

- **interpolate** (*bool*) – Enable smoothing (anti-aliasing) when drawing shape outlines. This produces a smoother (less-pixelated) outline of the shape.
- **name** (*str*) – Optional name of the stimuli for logging.
- **autoLog** (*bool*) – Enable auto-logging of events associated with this stimuli. Useful for debugging and to track timing when used in conjunction with *autoDraw*.
- **autoDraw** (*bool*) – Enable auto drawing. When *True*, the stimulus will be drawn every frame without the need to explicitly call the `draw()` method.
- **color** (array_like, *str*, *Color* or *None*) – Sets both the initial *lineColor* and *fillColor* of the shape.
- **lineRGB** (array_like, *Color* or *None*) – *Deprecated*. Please use *lineColor* and *fillColor*. These arguments may be removed in a future version.
- **fillRGB** (array_like, *Color* or *None*) – *Deprecated*. Please use *lineColor* and *fillColor*. These arguments may be removed in a future version.
- **lineColorSpace** (*str*) – Colorspace to use for the outline and fill. These change how the values passed to *lineColor* and *fillColor* are interpreted. *Deprecated*. Please use *colorSpace* to set both outline and fill colorspace. These arguments may be removed in a future version.
- **fillColorSpace** (*str*) – Colorspace to use for the outline and fill. These change how the values passed to *lineColor* and *fillColor* are interpreted. *Deprecated*. Please use *colorSpace* to set both outline and fill colorspace. These arguments may be removed in a future version.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

static _calcEquilateralVertices (*edges, radius=0.5*)

Get vertices for an equilateral shape with a given number of sides, will assume radius is 0.5 (relative) but can be manually specified

_calcPosRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`.

_calcSizeRendered ()

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`

_getDesiredRGB (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires `self.rgb`, `self.colorSpace` and `self.contrast`

_getPolyAsRendered ()

DEPRECATED. Return a list of vertices as rendered.

_selectWindow (*win*)

Switch drawing to the specified window. Calls the window's `_setCurrent()` method which handles the switch.

_set (*attrib, val, op="", log=None*)

DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

_tessellate (*newVertices*)

Set the `.vertices` and `.border` to new values, invoking tessellation.

_updateList ()

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

_updateVertices ()

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

autoDraw

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

autoLog

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property backColor

Alternative way of setting fillColor

property backColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property borderColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

closeShape

Should the last vertex be automatically connected to the first?

If you're using *Polygon*, *Circle* or *Rect*, *closeShape=True* is assumed and shouldn't be changed.

color

Set the color of the shape. Sets both *fillColor* and *lineColor* simultaneously if applicable.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'
```

(continues on next page)

(continued from previous page)

```
# Make it light green again
stim.color = (128, 255, 128)
```

contains (*x, y=None, units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such as a *Mouse*.**

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (*pos*), and orientation (*ori*) settings (and by the vertices for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw (*win=None, keepMatrix=False*)

Draw the stimulus in the relevant window.

You must call this method after every *win.flip()* if you want the stimulus to appear on that frame and then update the screen again.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use `setColor` if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

interpolate

If *True* the edge of the line will be anti-aliased.

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

lineWidth

Width of the line in **pixels**.

Operations supported.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. *visual.TextStim(win)* will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, *shapeContains.py*

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:


```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

setAutoDraw (*value*, *log=None*)

Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setBackRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setBorderColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setBorderRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Sets both the line and fill to be the same color.

setContrast (*newContrast*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setDKL (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setFillColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setFillRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*

setForeColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for *foreColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.

setForeRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOpacity (*newOpacity, operation="", log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos (*newPos, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color, operation="", log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize (*newSize, operation="", units=None, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setVertices (*value=None, operation="", log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property vertices

A list of lists or a numpy array (Nx2) specifying xy positions of each vertex, relative to the center of the field.

Assigning to vertices can be slow if there are many vertices.

Operations supported with *.setVertices()*.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.31 SimpleImageStim

class psychopy.visual.SimpleImageStim(*args, **kwargs)

A simple stimulus for loading images from a file and presenting at exactly the resolution and color in the file (subject to gamma correction if set).

Unlike the ImageStim, this type of stimulus cannot be rescaled, rotated or masked (although flipping horizontally or vertically is possible). Drawing will also tend to be marginally slower, because the image isn't preloaded to the graphics card. The slight advantage, however is that the stimulus will always be in its original aspect ratio, with no interpolation or other transformation, and it is slightly faster to load into PsychoPy.

9.3.32 Slider

Attributes

<i>Slider</i> (win[, ticks, labels, startValue, ...])	A class for obtaining ratings, e.g., on a 1-to-7 or categorical scale.
<i>Slider.getRating</i> ()	Get the current value of rating (or None if no response yet)
<i>Slider.getRT</i> ()	Get the RT for most recent rating (or None if no response yet)
<i>Slider.markerPos</i>	The position on the scale where the marker should be.
<i>Slider.setReadOnly</i> ([value, log])	When the rating scale is read only no responses can be made and the scale contrast is reduced
<i>Slider.contrast</i>	Set all elements of the Slider (labels, ticks, line) to a contrast
<i>Slider.style</i>	
<i>Slider.getHistory</i> ()	Return a list of the subject's history as (rating, time) tuples.
<i>Slider.getMouseResponses</i> ()	Instructs the rating scale to check for valid mouse responses.
<i>Slider.reset</i> ()	Resets the slider to its starting state (so that it can be restarted on each trial with a new stimulus)

Details

class `psychoPy.visual.Slider` (*win, ticks=1, 2, 3, 4, 5, labels=None, startValue=None, pos=0, 0, size=None, units=None, flip=False, ori=0, style='rating', styleTweaks=[], granularity=0, readOnly=False, labelColor='White', markerColor='Red', lineColor='White', colorSpace='rgb', opacity=None, font='Helvetica Bold', depth=0, name=None, labelHeight=None, labelWrapWidth=None, autoDraw=False, autoLog=True, color=False, fillColor=False, borderColor=False*)

A class for obtaining ratings, e.g., on a 1-to-7 or categorical scale.

A simpler alternative to `RatingScale`, to be customised with code rather than with arguments.

A `RatingScale` instance is a re-usable visual object having a `draw()` method, with customizable appearance and response options. `draw()` displays the rating scale, handles the subject's mouse or key responses, and updates the display. When the subject accepts a selection, `.noResponse` goes `False` (i.e., there is a response).

You can call the `getRating()` method anytime to get a rating, `getRT()` to get the decision time, or `getHistory()` to obtain the entire set of (rating, RT) pairs.

For other examples see Coder Demos -> stimuli -> ratingsNew.py.

Authors

- 2018: Jon Peirce

Parameters

- **win** (`psychoPy.visual.Window`) – Into which the scale will be rendered
- **ticks** (*list or tuple*) – A set of values for tick locations. If given a list of numbers then these determine the locations of the ticks (the first and last determine the endpoints and the rest are spaced according to their values between these endpoints).
- **labels** (*a list or tuple*) – The text to go with each tick (or spaced evenly across the ticks). If you give 3 labels but 5 tick locations then the end and middle ticks will be given labels. If the labels can't be distributed across the ticks then an error will be raised. If you want an uneven distribution you should include a list matching the length of ticks but with some values set to `None`
- **pos** (*XY pair (tuple, array or list)*) –
- **size** (*w,h pair (tuple, array or list)*) – The size for the scale defines the area taken up by the line and the ticks. This also controls whether the scale is horizontal or vertical.
- **units** (*the units to interpret the pos and size*) –
- **flip** (*bool*) – By default the labels will be below or left of the line. This puts them above (or right)
- **granularity** (*int or float*) – The smallest valid increments for the scale. 0 gives a continuous (e.g. “VAS”) scale. 1 gives a traditional likert scale. Something like 0.1 gives a limited fine-grained scale.
- / **color** (*labelColor*) – Color of the labels according to the color space
- / **fillColor** (*markerColor*) – Color of the marker according to the color space
- / **borderColor** (*lineColor*) – Color of the line and ticks according to the color space
- **font** (*font name*) –

- `autodraw` –
- `depth` –
- `name` –
- `autoLog` –

`__getHitboxParams ()`
Calculates hitbox size and pos from own size and pos

`__getLineParams ()`
Calculates location and size of the line based on own location and size

`__getMarkerParams ()`
Calculates location and size of marker based on own location and size

`__getTickParams ()`
Calculates the locations of the line, tickLines and labels from the rating info

`__granularRating (rating)`
Handle granularity for the rating

property `borderColor`

property `categorical`
(readonly) determines from labels and ticks whether the slider is categorical

contrast
Set all elements of the Slider (labels, ticks, line) to a contrast

Parameters `contrast` –

`draw ()`
Draw the Slider, with all its constituent elements on this frame

property `extent`
The distance from the leftmost point on the slider to the rightmost point, and from the highest point to the lowest.

property `fillColor`
Set the fill color for the shape.

property `flip`
1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

property `foreColor`
Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other *Color spaces***. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

getHistory()

Return a list of the subject's history as (rating, time) tuples.

The history can be retrieved at any time, allowing for continuous ratings to be obtained in real-time. Both numerical and categorical choices are stored automatically in the history.

getMarkerPos()

Get the current marker position (or None if no response yet)

getMouseResponses()

Instructs the rating scale to check for valid mouse responses.

This is usually done during the draw() method but can be done by the user as well at any point in time. The rating will be returned but will ALSO automatically be set as the current rating response.

While the mouse button is down we will alter self.markerPos but don't set a value for self.rating until button comes up

Returns

Return type A rating value or None

getRT()

Get the RT for most recent rating (or None if no response yet)

getRating()

Get the current value of rating (or None if no response yet)

property horiz

(readonly) determines from self.size whether the scale is horizontal

```

knownStyleTweaks = ['labels45', 'triangleMarker']
knownStyles = ['slider', 'rating', 'radio', 'scrollbar']
property labelColor
    Synonym of Slider.foreColor
property labelHeight
property labelWrapWidth
legacyStyleTweaks = ['whiteOnBlack']
legacyStyles = []
property markerColor
    Synonym of Slider.fillColor
markerPos
    The position on the scale where the marker should be. Note that this does not alter the value of the reported
    rating, only its visible display. Also note that this position is in scale units, not in coordinates
property opacity
property pos
property rating
recordRating (rating, rt=None, log=None)
    Sets the current rating value
reset ()
    Resets the slider to its starting state (so that it can be restarted on each trial with a new stimulus)
setMarkerPos (rating)
    Set the current marker position (or None if no response yet)
        Parameters rating (int or float) – The rating on the scale where we want to set the
            marker
setOpacity (value)
setReadOnly (value=True, log=None)
    When the rating scale is read only no responses can be made and the scale contrast is reduced
        Parameters
            • value (bool (True)) – The value to which we should set the readOnly flag
            • log (bool or None) – Force the autologging to occur or leave as default
property size
property style
styleTweaks
    Sets some predefined style tweaks or use these to create your own.
    If you fancy creating and including your own style tweaks that would be great!
        Parameters styleTweaks (list of strings) – Known style tweaks currently include:
            'triangleMarker': the marker is a triangle 'labels45': the text is rotated by 45 degrees
    Legacy style tweaks include:
        'whiteOnBlack': a sort of color-inverse rating scale

```

Legacy style tweaks will work if set in code, but are not exposed in Builder as they are redundant

Style tweaks can be combined in a list e.g. [*labels45*']

property units

property value

Synonymous with `.rating`

9.3.33 SphereStim

Attributes

<i>SphereStim</i> (win[, radius, subdiv, flipFaces, ...])	Class for drawing a UV sphere.
---	--------------------------------

Details

class psychopy.visual.**SphereStim**(win, radius=0.5, subdiv=32, 32, flipFaces=False, pos=0.0, 0.0, 0.0, ori=0.0, 0.0, 0.0, 1.0, color=0.0, 0.0, 0.0, colorSpace='rgb', contrast=1.0, opacity=1.0, useMaterial=None, name="", autoLog=True)

Class for drawing a UV sphere.

The resolution of the sphere mesh can be controlled by setting *sectors* and *stacks* which controls the number of latitudinal and longitudinal subdivisions, respectively. The radius of the sphere is defined by setting *radius* expressed in scene units (meters if using a perspective projection).

Calling the *draw* method will render the sphere to the current buffer. The render target (FBO or back buffer) must have a depth buffer attached to it for the object to be rendered correctly. Shading is used if the current window has light sources defined and lighting is enabled (by setting *useLights=True* before drawing the stimulus).

Warning: This class is experimental and may result in undefined behavior.

Examples

Creating a red sphere 1.5 meters away from the viewer with radius 0.25:

```
redSphere = SphereStim(win,
                        pos=(0., 0., -1.5),
                        radius=0.25,
                        color=(1, 0, 0))
```

Parameters

- **win** (~*psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.
- **radius** (*float*) – Radius of the sphere in scene units.
- **subdiv** (*array_like*) – Number of latitudinal and longitudinal subdivisions (*lat*, *long*) for the sphere mesh. The greater the number, the smoother the sphere will appear.

- **flipFaces** (*bool, optional*) – If *True*, normals and face windings will be set to point inward towards the center of the sphere. Texture coordinates will remain the same. Default is *False*.
- **pos** (*array_like*) – Position vector $[x, y, z]$ for the origin of the rigid body.
- **ori** (*array_like*) – Orientation quaternion $[x, y, z, w]$ where x, y, z are imaginary and w is real. If you prefer specifying rotations in axis-angle format, call *setOriAxisAngle* after initialization.
- **useMaterial** (*PhongMaterial, optional*) – Material to use. The material can be configured by accessing the *material* attribute after initialization. If not material is specified, the diffuse and ambient color of the shape will be set by *color*.
- **color** (*array_like*) – Diffuse and ambient color of the stimulus if *useMaterial* is not specified. Values are with respect to *colorSpace*.
- **colorSpace** (*str*) – Colorspace of *color* to use.
- **contrast** (*float*) – Contrast of the stimulus, value modulates the *color*.
- **opacity** (*float*) – Opacity of the stimulus ranging from 0.0 to 1.0. Note that transparent objects look best when rendered from farthest to nearest.
- **name** (*str*) – Name of this object for logging purposes.
- **autoLog** (*bool*) – Enable automatic logging on attribute changes.

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

__createVAO (*vertices, textureCoords, normals, faces*)

Create a vertex array object for handling vertex attribute data.

__getDesiredRGB (*rgb, colorSpace, contrast*)

Convert color to RGB while adding contrast. Requires *self.rgb*, *self.colorSpace* and *self.contrast*

__selectWindow (*win*)

Switch drawing to the specified window. Calls the window's *_setCurrent()* method which handles the switch.

__updateList ()

The user shouldn't need this method since it gets called after every call to *.set()* Chooses between using and not using shaders each call.

property anchor

property backColor

Alternative way of setting *fillColor*

property backColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property borderColor

property borderColorSpace

Deprecated, please use *colorSpace* to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

draw (*win=None*)

Draw the stimulus.

This should work for stimuli using a single VAO and material. More complex stimuli with multiple materials should override this method to correctly handle that case.

Parameters *win* (~*psychopy.visual.Window*) – Window this stimulus is associated with. Stimuli cannot be shared across windows unless they share the same context.

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use `obj._fillColor.rgb`

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as `True` to flip or `False` to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz
property flipVert
property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current `colorSpace`. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that `colorSpace='rgb'`:

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use `setColor` if you want to set color and `colorSpace` in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use `colorSpace` to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use `obj._foreColor.rgb`

Type DEPRECATED

getOri()

getOriAxisAngle (*degrees=True*)

Get the axis and angle of rotation for the 3D stimulus. Converts the orientation defined by the *ori* quaternion to and axis-angle representation.

Parameters **degrees** (*bool, optional*) – Specify `True` if *angle* is in degrees, or else it will be treated as radians. Default is `True`.

Returns Axis [*rx, ry, rz*] and angle.

Return type tuple

getPos()

getRayIntersectBounds (*rayOrig, rayDir*)

Get the point which a ray intersects the bounding box of this mesh.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [*x, y, z*].
- **rayDir** (*array_like*) – Direction vector of the ray [*x, y, z*], should be normalized.

Returns Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type tuple

getRayIntersectSphere (*rayOrig, rayDir*)

Get the point which a ray intersects the sphere.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [*x, y, z*].
- **rayDir** (*array_like*) – Direction vector of the ray [*x, y, z*], should be normalized.

Returns Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type tuple

property height

isVisible()

Check if the object is visible to the observer.

Test if a pose's bounding box or position falls outside of an eye's view frustum.

Poses can be assigned bounding boxes which enclose any 3D models associated with them. A model is not visible if all the corners of the bounding box fall outside the viewing frustum. Therefore any primitives (i.e. triangles) associated with the pose can be culled during rendering to reduce CPU/GPU workload.

Returns *True* if the object's bounding box is visible.

Return type `bool`

Examples

You can avoid running draw commands if the object is not visible by doing a visibility test first:

```
if myStim.isVisible():
    myStim.draw()
```

property `lineColor`

Alternative way of setting `borderColor`.

property `lineColorSpace`

Deprecated, please use `colorSpace` to set color space for the entire object

property `lineRGB`

Legacy property for setting the border color of a stimulus in RGB, instead use `obj._borderColor.rgb`

Type DEPRECATED

property `ori`

Orientation quaternion (X, Y, Z, W).

property `pos`

Position vector (X, Y, Z).

`setAnchor` (*value*, *log=None*)

`setBackColor` (*color*, *colorSpace=None*, *operation=""*, *log=None*)

`setBackRGB` (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set `obj._fillColor.rgb`

`setBorderColor` (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for `fillColor`, allows suppression of the log message, simultaneous `colorSpace` setting and calls update methods.

`setBorderRGB` (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set `obj._borderColor.rgb`

`setColor` (*color*, *colorSpace=None*, *operation=""*, *log=None*)

`setContrast` (*newContrast*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

`setDKL` (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the `color` attribute

`setFillColor` (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for `fillColor`, allows suppression of the log message, simultaneous `colorSpace` setting and calls update methods.

`setFillRGB` (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for fill RGB, instead set `obj._fillColor.rgb`

`setForeColor` (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for `foreColor`, allows suppression of the log message, simultaneous `colorSpace` setting and calls update methods.

setForeRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setLMS (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setLineColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

setLineRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*

setOri (*ori*)

setOriAxisAngle (*axis*, *angle*, *degrees=True*)

Set the orientation of the 3D stimulus using an *axis* and *angle*. This sets the quaternion at *ori*.

Parameters

- **axis** (*array_like*) – Axis of rotation [rx, ry, rz].
- **angle** (*float*) – Angle of rotation.
- **degrees** (*bool*, *optional*) – Specify True if *angle* is in degrees, or else it will be treated as radians. Default is True.

setPos (*pos*)

setRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

property size

property thePose

The pose of the rigid body. This is a class which has *pos* and *ori* attributes.

units

None, 'norm', 'cm', 'deg', 'degFlat', 'degFlatPos', or 'pix'

If None then the current units of the *Window* will be used. See *Units for the window and stimuli* for explanation of other options.

Note that when you change units, you don't change the stimulus parameters and it is likely to change appearance. Example:

```
# This stimulus is 20% wide and 50% tall with respect to window
stim = visual.PatchStim(win, units='norm', size=(0.2, 0.5)

# This stimulus is 0.2 degrees wide and 0.5 degrees tall.
stim.units = 'deg'
```

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a *textbox*

property vertices

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (*win1*, *win2* and *stim*):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.34 TextBox

Warning: `TextBox` is deprecated. Please use `TextBox2` instead which supports similar editable high-performance rendering of text but also supports non-monospaced fonts and a wider range of formatting and alignment options.

Attributes

`TextBox([window, text, font_name, bold, ...])` Similar to the `visual.TextStim` component, `TextBox` can be used to display text within a psychopy window.

The following `set_____()` attributes all have equivalent `get_____()` attributes:

<code>TextBox.setText(text_source)</code>	Set the text to be displayed within the Textbox.
<code>TextBox.setPosition(pos)</code>	Set the (x,y) position of the TextBox on the Monitor.
<code>TextBox.setHorzAlign(v)</code>	Specify how the horizontal (x) component of the TextBox position is to be interpreted.
<code>TextBox.setVertAlign(v)</code>	Specify how the vertical (y) component of the TextBox position is to be interpreted.
<code>TextBox.setHorzJust(v)</code>	Specify how text within the TextBox should be aligned horizontally.
<code>TextBox.setVertJust(v)</code>	Specify how text within the TextBox should be aligned vertically.
<code>TextBox.setFontColor(c)</code>	Set the color to use when drawing text glyphs within the TextBox.
<code>TextBox.setBorderColor(c)</code>	Set the color to use for the border of the textBox.
<code>TextBox.setBackgroundColor(c)</code>	Set the fill color used to fill the rectangular area of the TextBox stim.
<code>TextBox.setTextGridLineColor(c)</code>	Set the color used when drawing text grid lines.
<code>TextBox.setTextGridLineWidth(c)</code>	Set the stroke width (in pixels) to use for the text grid character bounding boxes.
<code>TextBox.setInterpolated(interpolate)</code>	Specify whether interpolation should be enabled for the TextBox when it is drawn.

continues on next page

Table 9.26 – continued from previous page

<code>TextBox.setOpacity(o)</code>	Sets the TextBox transparency level to use for color related attributes of the Textbox.
<code>TextBox.setAutoLog(v)</code>	Specify if changes to textBox attribute values should be logged automatically by PsychoPy.
<code>TextBox.draw()</code>	Draws the TextBox to the back buffer of the graphics card.

TextBox also provides the following read-only functions:

<code>TextBox.getSize()</code>	Return the width,height of the TextBox, using the unit type being used by the stimulus.
<code>TextBox.getName()</code>	Same as the GetLabel method.
<code>TextBox.getDisplayedText()</code>	Return the text that fits within the TextBox and therefore is actually seen. This is equal to:.
<code>TextBox.getValidStrokeWidths()</code>	Returns the stroke width range supported by the graphics card being used.
<code>TextBox.getLineSpacing()</code>	Return the additional spacing being applied between rows of text.
<code>TextBox.getGlyphPositionForTextIndex(char_index)</code>	FontManager provided char_index, which is the index of one character in
<code>TextBox.getTextGridCellPlacement()</code>	Returns a 3D numpy array containing position information for each text grid cell in the TextBox.

Helper Functions

getFontManager()

FontManager provides a simple API for finding and loading font files (.ttf) via the FreeType library.

The *FontManager* finds supported font files on the computer and initially creates a dictionary containing the information about available fonts. This can be used to quickly determine what font family names are available on the computer and what styles (bold, italic) are supported for each family.

This font information can then be used to create the resources necessary to display text using a given font family, style, size, color, and dpi.

The *FontManager* is currently used by the `psychopy.visual.TextBox` stim type. A user script can access the *FontManager* via:

```
font_mgr=visual.textbox.getFontManager()
```

Once a font of a given size and dpi has been created; it is cached by the *FontManager* and can be used by all *TextBox* instances created within the experiment.

Details

```
class psychopy.visual.TextBox (window=None, text='Default Test Text.', font_name=None, bold=False, italic=False, font_size=32, font_color=0, 0, 0, 1, dpi=72, line_spacing=0, line_spacing_units='pix', background_color=None, border_color=None, border_stroke_width=1, size=None, textgrid_shape=None, pos=0.0, 0.0, align_horz='center', align_vert='center', units='norm', grid_color=None, grid_stroke_width=1, color_space='rgb', opacity=1.0, grid_horz_justification='left', grid_vert_justification='top', autoLog=True, interpolate=False, name=None)
```

Similar to the visual.TextStim component, TextBox can be used to display text within a psychopy window. TextBox and TextStim each have different strengths and weaknesses. You should select the most appropriate text component type based on how it will be used within the experiment.

NOTE: As of PsychoPy 1.79, TextBox should be considered experimental. The two TextBox demo scripts provided have been tested on all PsychoPy supported OS's and run without exceptions. However there are very likely bugs in the existing TextBox code and the TextBox API will be further enhanced and improved (i.e. changed) over the next couple months.

TextBox Features

- Text character placement is very well defined, useful when the exact positioning of each letter needs to be known.
- The text string that is displayed can be changed (`setText()`) and drawn (`win.draw()`) **very** quickly. See the TextBox vs. TextStim comparison table for details.
- Built-in font manager; providing easy access to the font family names and styles that are available on the computer being used.
- TextBox is a composite stimulus type, with the following graphical elements, many of which can be changed to control many aspects of how the TextBox is displayed.:
 - TextBox Border / Outline
 - TextBox Fill Area
 - Text Grid Cell Lines
 - Text Glyphs
- When using 'rgb' or 'rgb255' color spaces, colors can be specified as a list/tuple of 3 elements (red, green, blue), or with four elements (red, green, blue, alpha) which allows different elements of the TextBox to use different opacity settings if desired. For colors that include the alpha channel value, it will be applied instead of the opacity setting of the TextBox, effectively overriding the stimulus defined opacity for that part of the textbox graphics. Colors that do not include an alpha channel use the opacity setting as normal.
- Text Line Spacing can be controlled.

Textbox Limitations

- Only Monospace Fonts are supported.
- TextBox component is not a completely **standard** psychopy visual stim and has the following functional difference:
 - TextBox attributes are never accessed directly; `get*` and `set*` methods are always used (this will be changed to use class properties in the future).
 - Setting an attribute of a TextBox only supports value replacement, (`textbox.setFontColor([1.0,1.0,1.0])`) and does not support specifying operators.

- Some key word arguments supported by other stimulus types in general, or by TextStim itself, are not supported by TextBox. See the TextBox class definition for the arguments that are supported.
- When a new font, style, and size are used it takes about 1 second to load and process the font. This is a one time delay for a given font name, style, and size. After first being loaded, the same font style can be used or re-applied to multiple TextBox components with no significant delay.
- Auto logging or auto drawing is not currently supported.

TextStim and TextBox Comparison:

Feature	TextBox	TextStim
Change text + redraw time [^]	1.513 msec	28.537 msec
No change + redraw time [^]	0.240 msec	0.931 msec
Initial Creation time [^]	0.927 msec	0.194 msec
MonoSpace Font Support	Yes	Yes
Non MonoSpace Font Support	No	Yes
Adjustable Line Spacing	Yes	No
Precise Text Pos. Info	Yes	No
Auto logging Support	No	Yes
Rotation Support	No	Yes
Word Wrapping Support	Yes	Yes

[^] **Times are in msec.usec format. Tested using the `textstim_vs_textbox.py` demo script provided with the PsychoPy distribution.** Results are dependent on text length, video card, and OS. Displayed results are based on 120 character string with an average of 24 words. Test computer used Windows 7 64 bit, PsychoPy 1.79, with a i7 3.4 Ghz CPU, 8 GB RAM, and NVIDIA 480 GTX 2GB graphics card.

Example:

```

from psychopy import visual

win=visual.Window(...)

# A Textbox stim that will look similar to a TextStim component

textstimlike=visual.TextBox(
    window=win,
    text="This textbox looks most like a textstim.",
    font_size=18,
    font_color=[-1,-1,1],
    color_space='rgb',
    size=(1.8, .1),
    pos=(0.0, .5),
    units='norm')

# A Textbox stim that uses more of the supported graphical features
#
textboxloaded=visual.TextBox(
    window=win
    text='TextBox showing all supported graphical elements',
    font_size=32,
    font_color=[1,1,1],
    border_color=[-1,-1,1], # draw a blue border around stim
    border_stroke_width=4, # border width of 4 pix.
    background_color=[-1,-1,-1], # fill the stim background

```

(continues on next page)

(continued from previous page)

```

grid_color=[1,-1,-1,0.5], # draw a red line around each
                        # possible letter area,
                        # 50% transparent
grid_stroke_width=1, # with a width of 1 pix
textgrid_shape=[20,2], # specify area of text box
                  # by the number of cols x
                  # number of rows of text to support
                  # instead of by a screen
                  # units width x height.

pos=(0.0,-.5),
# If the text string length < num rows * num cols in
# textgrid_shape, how should text be justified?
#
grid_horz_justification='center',
grid_vert_justification='center')

textstimlike.draw()
textboxloaded.draw()
win.flip()

```

draw()

Draws the TextBox to the back buffer of the graphics card. Then call win.flip() to display the changes drawn. If draw() is not called prior to a call to win.flip(), the textBox will not be displayed for that retrace.

getAutoLog()

Indicates if changes to textBox attribute values should be logged automatically by PsychoPy. *Currently not supported by TextBox.*

getBackgroundColor()

Get the color used to fill the rectangular area of the TextBox stim. All other graphical elements of the TextBox are drawn on top of the background.

getBorderColor()

A border can be drawn around the perimeter of the TextBox. This method sets the color of that border.

getBorderWidth()

Get the stroke width of the optional TextBox area outline. This is always given in pixel units.

getColorSpace()

Returns the psychopy color space used when specifying colors for the TextBox. Supported values are:

- 'rgb'
- 'rgb255'
- 'norm'
- hex (implicit)
- html name (implicit)

See the Color Space section of the PsychoPy docs for details.

getDisplayText()

Return the text that fits within the TextBox and therefore is actually seen. This is equal to:

```

text_length=len(self.getText())
cols,rows=self.getTextGridShape()

displayed_text=self.getText()[0:min(text_length,rows*cols)]

```

getFontColor ()

Return the color used when drawing text glyphs.

getFontSize ()

getGlyphPositionForTextIndex (char_index)

For the provided **char_index**, which is the index of one character in the current text being displayed by the TextBox (`getDisplayedText()`), return the bounding box position, width, and height for the associated glyph drawn to the screen. This factors in the glyphs position within the textgrid cell it is being drawn in, so the returned bounding box is for the actual glyph itself, not the textgrid cell. For textgrid cell placement information, see the `getTextGridCellPlacement()` method.

The glyph position for the given text index is returned as a tuple (x,y,width,height), where x,y is the top left hand corner of the bounding box.

Special Cases:

- If the index provided is out of bounds for the currently displayed text, None is returned.
- For u' ' (space) characters, the full textgrid cell bounding box is returned.
- For u'

' (new line) characters, the textgrid cell bounding box is returned, but with the box width set to 0.

getHorzAlign ()

Return what textbox x position should be interpreted as. Valid options are 'left', 'center', or 'right' .

getHorzJust ()

Return how text should laid out horizontally when the number of columns of each text grid row is greater than the number needed to display the text for that text row.

getInterpolated ()

Returns whether interpolation is enabled for the TextBox when it is drawn. When True, GL_LINE_SMOOTH and GL_POLYGON_SMOOTH are enabled within OpenGL; otherwise they are disabled.

getLabel ()

Return the label / name assigned to the textbox. This does not impact how the stimulus looks when drawn, and instead is used for internal purposes only.

getLineSpacing ()

Return the additional spacing being applied between rows of text. The value is in units specified by the textbox `getUnits()` method.

getName ()

Same as the GetLabel method.

getOpacity ()

Get the default TextBox transparency level used for color related attributes. 0.0 equals fully transparent, 1.0 equals fully opaque.

getPosition ()

Return the x,y position of the textbox, in `getUnitType()` coord space.

getSize ()

Return the width,height of the TextBox, using the unit type being used by the stimulus.

getText ()

Return the text to display.

getTextGridCellForCharIndex (char_index)

getTextGridCellPlacement ()

Returns a 3D numpy array containing position information for each text grid cell in the TextBox. The array has the shape (*num_cols*, *num_rows*, *cell_bounds*), where *num_cols* is the number of *textgrid* columns in the TextBox. *num_rows* is the number of *textgrid* rows in the *TextBox*. *cell_bounds* is a 4 element array containing the (x pos, y pos, width, height) data for the given cell. Position fields are for the top left hand corner of the cell box. Column and Row indices start at 0.

To get the shape of the textgrid in terms of columns and rows, use:

```
cell_pos_array=textbox.getTextGridCellPlacement ()
col_row_count=cell_pos_array.shape [:2]
```

To access the position, width, and height for textgrid cell at column 0 and row 0 (so the top left cell in the textgrid):

```
cell00=cell_pos_array [0,0,:]
```

For the cell at col 3, row 1 (so 4th cell on second row):

```
cell41=cell_pos_array [4,1,:]
```

getTextGridLineColor ()

Return the color used when drawing the outline of the text grid cells. Each letter displayed in a TextBox populates one of the text cells defined by the shape of the TextBox text grid. Color value must be valid for the color space being used by the TextBox.

A value of None indicates drawing of the textgrid lines is disabled.

getTextGridLineWidth ()

Return the stroke width (in pixels) of the optional lines drawn around the text grid cell areas.

getUnitType ()

Returns which of the psychopy coordinate systems are used by the TextBox. Position and size related attributes must be specified relative to the unit type being used. Valid options are:

- pix
- norm
- cm

getValidStrokeWidths ()

Returns the stroke width range supported by the graphics card being used. If the TextBox is Interpolated, a tuple is returned using float values, with the following structure:

```
((min_line_width, max_line_width), line_width_granularity)
```

If Interpolation is disabled for the TextBox, the returned tuple elements are int values, with the following structure:

```
(min_line_width, max_line_width)
```

getVertAlign ()

Return what textbox y position should be interpreted as. Valid options are 'top', 'center', or 'bottom'.

getVertJust ()

Return how text should be laid out vertically when the number of text grid rows is greater than the number needed to display the current text

getWindow ()

Returns the psychopy window that the textBox is associated with.

setAutoLog (*v*)

Specify if changes to textBox attribute values should be logged automatically by PsychoPy. True enables auto logging; False disables it. *Currently not supported by TextBox.*

setBackgroundcolor (*c*)

Set the fill color used to fill the rectangular area of the TextBox stim. Color value must be valid for the color space being used by the TextBox.

A value of None will disable drawing of the TextBox background.

setbordercolor (*c*)

Set the color to use for the border of the textBox. The TextBox border is a rectangular outline drawn around the edges of the TextBox stim. Color value must be valid for the color space being used by the TextBox.

A value of None will disable drawing of the border.

setborderwidth (*c*)

Set the stroke width (in pixels) to use for the border of the TextBox stim. Border values must be within the range of stroke widths supported by the OpenGL driver used by the graphics. Setting the width outside the valid range will result in the stroke width being clamped to the nearest end of the valid range.

Use the `TextBox.getValidStrokeWidths()` to access the minimum - maximum range of valid line widths.

setFontcolor (*c*)

Set the color to use when drawing text glyphs within the TextBox. Color value must be valid for the color space being used by the TextBox. For 'rgb', 'rgb255', and 'norm' based colors, three or four element lists are valid. Three element colors use the `TextBox.getOpacity()` value to determine the alpha channel for the color. Four element colors use the value of the fourth element to set the alpha value for the color.

setHorzAlign (*v*)

Specify how the horizontal (x) component of the TextBox position is to be interpreted. left = x position is the left edge, right = x position is the right edge x position, and center = the x position is used to center the stim horizontally.

setHorzJust (*v*)

Specify how text within the TextBox should be aligned horizontally. For example, if a text grid has 10 columns, and the text being displayed is 6 characters in length, the horizontal justification determines if the text should be drawn starting at the left of the text columns (left), or should be centered on the columns ('center', in this example there would be two empty text cells to the left and right of the text.), or should be drawn such that the last letter of text is drawn in the last column of the text row ('right').

setInterpolated (*interpolate*)

Specify whether interpolation should be enabled for the TextBox when it is drawn. When `interpolate == True`, `GL_LINE_SMOOTH` and `GL_POLYGON_SMOOTH` are enabled within OpenGL. When `interpolate` is set to `False`, `GL_POLYGON_SMOOTH` and `GL_LINE_SMOOTH` are disabled.

setOpacity (*o*)

Sets the TextBox transparency level to use for color related attributes of the Textbox. 0.0 equals fully transparent, 1.0 equals fully opaque.

If opacity is set to None, it is assumed to have a default value of 1.0.

When a color is defined with a 4th element in the colors element list, then this opacity value is ignored and the alpha value provided in the color itself is used for that TextGrid element instead.

setPosition (*pos*)

Set the (x,y) position of the TextBox on the Monitor. The position must be given using the unit coord type used by the stim.

The TextBox position is interpreted differently depending on the Horizontal and Vertical Alignment settings of the stim. See `getHorzAlignment()` and `getVertAlignment()` for more information.

For example, if the TextBox alignment is specified as left, top, then the position specifies the top left hand corner of where the stim will be drawn. An alignment of bottom,right indicates that the position value will define where the bottom right corner of the TextBox will be drawn. A horz., vert. alignment of center, center will place the center of the TextBox at pos.

setText (*text_source*)

Set the text to be displayed within the Textbox.

Note that once a TextBox has been created, the number of character rows and columns is static. To change the size of a TextBox, a new TextBox stim must be created to replace the current Textbox stim. Therefore ensure that the textbox is large enough to display the largest length string to be presented in the TextBox. Characters that do not fit within the TextBox will not be displayed.

Color value must be valid for the color space being used by the TextBox.

setTextGridLineColor (*c*)

Set the color used when drawing text grid lines. These are lines that can be drawn which mark the bounding box for each character within the TextBox text grid. Color value must be valid for the color space being used by the TextBox.

Provide a value of None to disable drawing of textgrid lines.

setTextGridLineWidth (*c*)

Set the stroke width (in pixels) to use for the text grid character bounding boxes. Border values must be within the range of stroke widths supported by the OpenGL driver used by the computer graphics card. Setting the width outside the valid range will result in the stroke width being clamped to the nearest end of the valid range.

Use the `TextBox.getGLLineRanges()` to access a dict containing some OpenGL parameters which provide the minimum, maximum, and resolution of valid line widths.

setVertAlign (*v*)

Specify how the vertical (y) component of the TextBox position is to be interpreted. top = y position is the top edge, bottom = y position is the bottom edge y position, and center = the y position is used to center the stim vertically.

setVertJust (*v*)

Specify how text within the TextBox should be aligned vertically. For example, if a text grid has 3 rows for text, and the text being displayed all fits on one row, the vertical justification determines if the text should be draw on the top row of the text grid (top), or should be centered on the rows ('center', in this example there would be one row above and below the row used to draw the text), or should be drawn on the last row of the text grid, ('bottom').

9.3.35 TextBox2

Attributes

TextBox2(win, text[, font, pos, units, ...])

param win

The following `set_____()` attributes all have equivalent `get_____()` attributes:

<code>TextBox2.text</code>	
<code>TextBox2.alignment</code>	
<code>TextBox2.hasFocus</code>	
<code>TextBox2.overlaps(polygon[, tight])</code>	Returns <i>True</i> if this stimulus intersects another one.
<code>TextBox2.contains(x[, y, units, tight])</code>	Returns True if a point x,y is inside the stimulus' border.
<code>TextBox2.clear()</code>	Resets the TextBox2 to a blank string
<code>TextBox2.reset()</code>	Resets the TextBox2 to hold whatever it was given on initialisation
<code>TextBox2.font</code>	
<code>TextBox2.height</code>	
<code>TextBox2.anchor</code>	
<code>TextBox2.editable</code>	Determines whether or not the TextBox2 instance can receive typed text
<code>TextBox2.padding</code>	
<code>TextBox2.size</code>	The (requested) size of the TextBox (w,h) in whatever units the stimulus is using
<code>TextBox2.pos</code>	The position of the center of the TextBox in the stimulus <i>units</i>
<code>TextBox2.units</code>	
<code>TextBox2.draw()</code>	Draw the text to the back buffer

Details

class psychopy.visual.**TextBox2** (*win, text, font='Open Sans', pos=0, 0, units=None, letterHeight=None, size=None, color=1.0, 1.0, 1.0, colorSpace='rgb', fillColor=None, fillColorSpace=None, borderWidth=2, borderColor=None, borderColorSpace=None, contrast=1, opacity=None, bold=False, italic=False, lineSpacing=None, padding=None, anchor='center', alignment='left', flipHoriz=False, flipVert=False, languageStyle='LTR', editable=False, lineBreaking='default', name='', autoLog=None, autoDraw=False, onTextCallback=None*)

Parameters

- **win** –
- **text** –
- **font** –
- **pos** –
- **units** –
- **letterHeight** –
- **size** (Specifying *None* gets the default size for this type of *unit*.) – Specifying [None, None] gets a TextBox that's expandable in both dimensions. Specifying [0.75, None] gets a textbox that expands in the length but fixed at 0.75 units in the width
- **color** –
- **colorSpace** –
- **contrast** –

- **opacity** –
- **bold** –
- **italic** –
- **lineSpacing** –
- **padding** –
- **anchor** –
- **alignment** –
- **fillColor** –
- **borderWidth** –
- **borderColor** –
- **flipHoriz** –
- **flipVert** –
- **editable** –
- **lineBreaking** (*Specifying 'default', text will be broken at a set of*) – characters defined in the module. Specifying 'uax14', text will be broken in accordance with UAX#14 (Unicode Line Breaking Algorithm).
- **name** –
- **autoLog** –

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`

`_getDesiredRGB(rgb, colorSpace, contrast)`

Convert color to RGB while adding contrast. Requires `self.rgb`, `self.colorSpace` and `self.contrast`

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_layout()`

Layout the text, calculating the vertex locations

`_onCursorKeys(key)`

Called by the window when `cursor/del/backspace...` are received

`_onText(chr)`

Called by the window when characters are received

`_selectWindow(win)`

Switch drawing to the specified window. Calls the window's `_setCurrent()` method which handles the switch.

`_set(attr, val, op="", log=None)`

DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

_updateList ()

The user shouldn't need this method since it gets called after every call to .set() Chooses between using and not using shaders each call.

_updateVertices ()

Sets Stim.verticesPix and ._borderPix from pos, size, ori, flipVert, flipHoriz

addCharAtCaret (*char*)

Allows a character to be added programmatically at the current caret

property alignment

property anchor

autoDraw

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

autoLog

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property backColor

Alternative way of setting fillColor

property backColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property backRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property borderColor

property borderColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property borderRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

clear ()

Resets the TextBox2 to a blank string

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                       color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains (*x, y=None, units=None, tight=False*)
Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such** as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (*pos*), and orientation (*ori*) settings (and by the vertices for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

deleteCaretLeft ()

Deletes 1 character to the left of the caret

deleteCaretRight ()

Deletes 1 character to the right of the caret

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw()

Draw the text to the back buffer

property editable

Determines whether or not the TextBox2 instance can receive typed text

property fillColor

Set the fill color for the shape.

property fillColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property fillRGB

Legacy property for setting the fill color of a stimulus in RGB, instead use *obj._fillColor.rgb*

Type DEPRECATED

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

font

property fontMGR

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other Color spaces.** For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current colorSpace. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
                 # space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property foreColorSpace

Deprecated, please use colorSpace to set color space for the entire object.

property foreRGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

getText ()

Returns the current text in the box, including formatting tokens.

getVisibleText ()

Returns the current visible text in the box

property hasFocus

property height

property languageStyle

How is text laid out? Left to right (LTR), right to left (RTL) or using Arabic layout rules?

property letterHeight

property letterHeightPix

Convenience function to get self._letterHeight.pix and be guaranteed a return that is a single integer

property lineColor

Alternative way of setting *borderColor*.

property lineColorSpace

Deprecated, please use colorSpace to set color space for the entire object

property lineRGB

Legacy property for setting the border color of a stimulus in RGB, instead use *obj._borderColor.rgb*

Type DEPRECATED

property lineSpacing

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon, tight=False*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

Parameters

See coder demo, shapeContains.py

property padding

property palette

Describes the current visual properties of the TextBox in a dict

property pallette

Describes the current visual properties of the TextBox in a dict

property pos

The position of the center of the TextBox in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
myTextbox._pos.pix
```

reset ()

Resets the TextBox2 to hold **whatever it was given on initialisation**

setAnchor (*value, log=None*)

setAutoDraw (*value, log=None*)

Sets autoDraw. Usually you can use 'stim.attribute = value' syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True, log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setBackColor (*color, colorSpace=None, operation="", log=None*)

- setBackRGB** (*color*, *operation=""*, *log=None*)
 DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*
- setBorderColor** (*color*, *colorSpace=None*, *operation=""*, *log=None*)
 Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.
- setBorderRGB** (*color*, *operation=""*, *log=None*)
 DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*
- setColor** (*color*, *colorSpace=None*, *operation=""*, *log=None*)
- setContrast** (*newContrast*, *operation=""*, *log=None*)
 Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message
- setDKL** (*color*, *operation=""*)
 DEPRECATED since v1.60.05: Please use the *color* attribute
- setDepth** (*newDepth*, *operation=""*, *log=None*)
 Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message
- setFillColor** (*color*, *colorSpace=None*, *operation=""*, *log=None*)
 Hard setter for *fillColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.
- setFillRGB** (*color*, *operation=""*, *log=None*)
 DEPRECATED: Legacy setter for fill RGB, instead set *obj._fillColor.rgb*
- setFont** (*font*, *log=None*)
 Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.
- setForeColor** (*color*, *colorSpace=None*, *operation=""*, *log=None*)
 Hard setter for *foreColor*, allows suppression of the log message, simultaneous *colorSpace* setting and calls update methods.
- setForeRGB** (*color*, *operation=""*, *log=None*)
 DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*
- setHeight** (*height*, *log=None*)
 Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.
- setLMS** (*color*, *operation=""*)
 DEPRECATED since v1.60.05: Please use the *color* attribute
- setLineColor** (*color*, *colorSpace=None*, *operation=""*, *log=None*)
- setLineRGB** (*color*, *operation=""*, *log=None*)
 DEPRECATED: Legacy setter for border RGB, instead set *obj._borderColor.rgb*
- setOpacity** (*newOpacity*, *operation=""*, *log=None*)
 Hard setter for opacity, allows the suppression of log messages and calls the update method
- setOri** (*newOri*, *operation=""*, *log=None*)
 Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message
- setPos** (*newPos*, *operation=""*, *log=None*)
 Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setRGB (*color, operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize (*newSize, operation=""*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setText (*text=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

property size

The (requested) size of the TextBox (w,h) in whatever units the stimulus is using

This determines the outer extent of the area.

If the width is set to None then the text will continue extending and not wrap. If the height is set to None then the text will continue to grow downwards as needed.

property startText

In v2022.1.4, *.startText* was replaced by *.placeholder* for consistency with PsychoJS. The two attributes are fully interchangeable.

property text

property units

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property visibleText

Returns the current visible text in the box

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:


```
stim.draw(win1)
stim.draw(win2)
```

9.3.36 TextStim

```
class psychopy.visual.TextStim(win, text='Hello World', font="", pos=0.0, 0.0, depth=0,
    rgb=None, color=1.0, 1.0, 1.0, colorSpace='rgb', opacity=1.0,
    contrast=1.0, units="", ori=0.0, height=None, antialias=True,
    bold=False, italic=False, alignHoriz=None, alignVert=None,
    alignText='center', anchorHoriz='center', anchorVert='center',
    fontFiles=(), wrapWidth=None, flipHoriz=False, flipVert=False,
    languageStyle='LTR', name=None, autoLog=None, auto-
    Draw=False)
```

Class of text stimuli to be displayed in a [Window](#)

Performance OBS: in general, TextStim is slower than many other visual stimuli, i.e. it takes longer to change some attributes. In general, it's the attributes that affect the shapes of the letters: `text`, `height`, `font`, `bold` etc. These make the next `.draw()` slower because that sets the text again. You can make the `draw()` quick by calling re-setting the text (`myTextStim.text = myTextStim.text`) when you've changed the parameters.

In general, other attributes which merely affect the presentation of unchanged shapes are as fast as usual. This includes `pos`, `opacity` etc.

The following attribute can only be set at initialization (see further down for a list of attributes which can be changed after initialization):

languageStyle Apply settings to correctly display content from some languages that are written right-to-left. Currently there are three (case- insensitive) values for this parameter:

- **'LTR'** is the default, for typical left-to-right, Latin-style languages.
- **'RTL'** will correctly display text in right-to-left languages such as Hebrew. By applying the bidirectional algorithm, it allows mixing portions of left-to-right content (such as numbers or Latin script) within the string.
- **'Arabic'** applies the bidirectional algorithm but additionally will `_reshape_` Arabic characters so they appear in the cursive, linked form that depends on neighbouring characters, rather than in their isolated form. May also be applied in other scripts, such as Farsi or Urdu, that use Arabic-style alphabets.

Parameters

property RGB

Legacy property for setting the foreground color of a stimulus in RGB, instead use `obj._foreColor.rgb`

Type DEPRECATED

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`

`_getDesiredRGB(rgb, colorSpace, contrast)`

Convert color to RGB while adding contrast. Requires `self.rgb`, `self.colorSpace` and `self.contrast`

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_selectWindow(win)`

Switch drawing to the specified window. Calls the window's `_setCurrent()` method which handles the switch.

`_set(attrib, val, op="", log=None)`

DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

`_setTextShaders(value=None)`

Set the text to be rendered using the current font

`_updateList()`

The user shouldn't need this method since it gets called after every call to `.set()` Chooses between using and not using shaders each call.

`_updateListShaders()`

Only used with pygame text - pyglet handles all from the `draw()`

`_updateVertices()`

Sets `Stim.verticesPix` and `._borderPix` from `pos`, `size`, `ori`, `flipVert`, `flipHoriz`

`alignHoriz`

Deprecated in PsychoPy 3.3. Use `alignText` and `anchorHoriz` instead

`alignText`

Aligns the text content within the bounding box ('left', 'right' or 'center') See also `anchorX` to set alignment of the box itself relative to `pos`

`alignVert`

Deprecated in PsychoPy 3.3. Use `anchorVert`

`anchorHoriz`

The horizontal alignment ('left', 'right' or 'center')

`anchorVert`

The vertical alignment ('top', 'bottom' or 'center') of the box relative to the text `pos`.

`antialias`

Allow antialiasing the text (True or False). Sets `text`, `slow`.

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

`bold`

Make the text bold (True, False) (better to use a bold font name).

`property boundingBox`

(read only) attribute representing the bounding box of the text (w,h). This differs from `width` in that the width represents the width of the margins, which might differ from the width of the text within them.

NOTE: currently always returns the size in pixels (this will change to return in stimulus units)

property color

Alternative way of setting *foreColor*.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

contains (*x, y=None, units=None*)

Returns True if a point x,y is inside the stimulus' border.

Can accept variety of input options:

- two separate args, x and y
- one arg (list, tuple or array) containing two vals (x,y)
- **an object with a getPos() method that returns x,y, such** as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no *.border*. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the size, position (*pos*), and orientation (*ori*) settings (and by the vertices for shape stimuli).

See Coder demos: *shapeContains.py* See Coder demos: *shapeContains.py*

property contrast

A value that is simply multiplied by the color.

Value should be: a float between -1 (negative) and 1 (unchanged). *Operations* supported.

Set the contrast of the stimulus, i.e. scales how far the stimulus deviates from the middle grey. You can also use the stimulus *opacity* to control contrast, but that cannot be negative.

Examples:

```
stim.contrast = 1.0 # unchanged contrast
stim.contrast = 0.5 # decrease contrast
stim.contrast = 0.0 # uniform, no contrast
stim.contrast = -0.5 # slightly inverted
stim.contrast = -1.0 # totally inverted
```

Setting contrast outside range -1 to 1 is permitted, but may produce strange results if color values exceeds the monitor limits.:

```
stim.contrast = 1.2 # increases contrast
stim.contrast = -1.2 # inverts with increased contrast
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw (*win=None*)

Draw the stimulus in its relevant window. You must call this method after every `MyWin.flip()` if you want the stimulus to appear on that frame and then update the screen again.

If *win* is specified then override the normal window of this stimulus.

property flip

1x2 array for flipping vertices along each axis; set as True to flip or False to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (`._flip`) will give an array of 1s and -1s with which to multiply vertices.

flipHoriz

If set to True then the text will be flipped left-to-right. The flip is relative to the original, not relative to the current state.

flipVert

If set to True then the text will be flipped top-to-bottom. The flip is relative to the original, not relative to the current state.

font

String. Set the font to be used for text rendering. *font* should be a string specifying the name of the font (in system resources).

fontFiles

A list of additional files if the font is not in the standard system location (include the full path).

OBS: fonts are added every time this value is set. Previous are not deleted.

E.g.:

```
stim.fontFiles = ['SpringRage.ttf'] # load file(s)
stim.font = 'SpringRage' # set to font
```

property foreColor

Foreground color of the stimulus

Value should be one of:

- string: to specify a *Colors by name*. Any of the standard html/X11 *color names* <http://www.w3schools.com/html/html_colornames.asp> can be used.
- *Colors by hex value*
- **numerically: (scalar or triplet) for DKL, RGB or other *Color spaces***. For these, *operations* are supported.

When color is specified using numbers, it is interpreted with respect to the stimulus' current `colorSpace`. If color is given as a single value (scalar) then this will be applied to all 3 channels.

Examples

For whatever stim you have:

```
stim.color = 'white'
stim.color = 'RoyalBlue' # (the case is actually ignored)
stim.color = '#DDA0DD' # DDA0DD is hexadecimal for plum
stim.color = [1.0, -1.0, -1.0] # if stim.colorSpace='rgb':
# a red color in rgb space
stim.color = [0.0, 45.0, 1.0] # if stim.colorSpace='dkl':
# DKL space with elev=0, azimuth=45
stim.color = [0, 0, 255] # if stim.colorSpace='rgb255':
# a blue stimulus using rgb255 space
stim.color = 255 # interpreted as (255, 255, 255)
# which is white in rgb255.
```

Operations work as normal for all numeric colorSpaces (e.g. 'rgb', 'hsv' and 'rgb255') but not for strings, like named and hex. For example, assuming that colorSpace='rgb':

```
stim.color += [1, 1, 1] # increment all guns by 1 value
stim.color *= -1 # multiply the color by -1 (which in this
# space inverts the contrast)
stim.color *= [0.5, 0, 1] # decrease red, remove green, keep blue
```

You can use *setColor* if you want to set color and colorSpace in one line. These two are equivalent:

```
stim.setColor((0, 128, 255), 'rgb255')
# ... is equivalent to
stim.colorSpace = 'rgb255'
stim.color = (0, 128, 255)
```

property **foreColorSpace**

Deprecated, please use colorSpace to set color space for the entire object.

property **foreRGB**

Legacy property for setting the foreground color of a stimulus in RGB, instead use *obj._foreColor.rgb*

Type DEPRECATED

height

The height of the letters (Float/int or None = set default).

Height includes the entire box that surrounds the letters in the font. The width of the letters is then defined by the font.

Operations supported.

italic

True/False. Make the text italic (better to use a italic font name).

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. visual.TextStim(win) will be called “unnamed TextStim” in the logs.

property **opacity**

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, shapeContains.py

property pos

The position of the center of the stimulus in the stimulus *units*

value should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

property posPix

This determines the coordinates in pixels of the position for the current stimulus, accounting for pos and units. This property should automatically update if *pos* is changed

setAutoDraw (*value*, *log=None*)

Sets autoDraw. Usually you can use ‘stim.attribute = value’ syntax instead, but use this method to suppress the log message.

setAutoLog (*value=True*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setContrast (*newContrast*, *operation=""*, *log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setDKL (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setDepth (*newDepth*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setFlip (*direction*, *log=None*)

(used by Builder to simplify the dialog)

setFlipHoriz (*newVal=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setFlipVert (*newVal=True*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setFont (*font*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setForeColor (*color*, *colorSpace=None*, *operation=""*, *log=None*)

Hard setter for foreColor, allows suppression of the log message, simultaneous colorSpace setting and calls update methods.

setForeRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setHeight (*height*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setLMS (*color*, *operation=""*)

DEPRECATED since v1.60.05: Please use the *color* attribute

setOpacity (*newOpacity*, *operation=""*, *log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setPos (*newPos*, *operation=""*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

setRGB (*color*, *operation=""*, *log=None*)

DEPRECATED: Legacy setter for foreground RGB, instead set *obj._foreColor.rgb*

setSize (*newSize*, *operation=""*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setText (*text=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message.

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

text

The text to be rendered. Use `\n` to make new lines.

Issues: May be slow, and pyglet has a memory leak when setting text. For these reasons, this function checks so that it only updates the text if it has changed. So scripts can safely set the text on every frame, with no need to check if it has actually altered.

updateColors ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (win1, win2 and stim):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

wrapWidth

Int/float or None (set default). The width the text should run before wrapping.

Operations supported.

9.3.37 VlcMovieStim

A stimulus class for playing movies (mpeg, avi, etc...) in using a local installation of VLC media player (<https://www.videolan.org/>).

Requires version 3.0.11115 of `python-vlc` on Windows. Other platforms (MacOS and Linux) may use a newer version.

Attributes

<code>VlcMovieStim(win[, filename, units, size, ...])</code>	A stimulus class for playing movies in various formats (mpeg, avi, etc...) in PsychoPy using the VLC media player as a decoder.
<code>VlcMovieStim.win</code>	The <i>Window</i> object in which the stimulus will be rendered by default.
<code>VlcMovieStim.units</code>	
<code>VlcMovieStim.pos</code>	The position of the center of the stimulus in the stimulus <i>units</i>
<code>VlcMovieStim.ori</code>	The orientation of the stimulus (in degrees).
<code>VlcMovieStim.size</code>	The size (width, height) of the stimulus in the stimulus <i>units</i>
<code>VlcMovieStim.opacity</code>	Determines how visible the stimulus is relative to background.
<code>VlcMovieStim.name</code>	The name (<i>str</i>) of the object to be using during logged messages about this stim.
<code>VlcMovieStim.autoLog</code>	Whether every change in this stimulus should be auto logged.
<code>VlcMovieStim.draw([win])</code>	Draw the current frame to a particular <i>Window</i> (or to the default win for this object if not specified).
<code>VlcMovieStim.autoDraw</code>	Determines whether the stimulus should be automatically drawn on every frame flip.
<code>VlcMovieStim.setMovie(filename[, log])</code>	See <code>~MovieStim.loadMovie</code> (the functions are identical).
<code>VlcMovieStim.loadMovie(filename[, log])</code>	Load a movie from file
<code>VlcMovieStim.isPlaying</code>	<i>True</i> if the video is presently playing (<i>bool</i>).
<code>VlcMovieStim.isNotStarted</code>	<i>True</i> if the video has not been started yet (<i>bool</i>).
<code>VlcMovieStim.isStopped</code>	<i>True</i> if the video is stopped (<i>bool</i>).
<code>VlcMovieStim.isPaused</code>	<i>True</i> if the video is presently paused (<i>bool</i>).
<code>VlcMovieStim.isFinished</code>	<i>True</i> if the video is finished (<i>bool</i>).
<code>VlcMovieStim.play([log])</code>	Start or continue a paused movie from current position.
<code>VlcMovieStim.pause([log])</code>	Pause the current point in the movie.
<code>VlcMovieStim.stop([log])</code>	Stop the current point in the movie (sound will stop, current frame will not advance).
<code>VlcMovieStim.seek(timestamp[, log])</code>	Seek to a particular timestamp in the movie.
<code>VlcMovieStim.rewind([seconds])</code>	Rewind the video.
<code>VlcMovieStim.fastForward([seconds])</code>	Fast-forward the video.
<code>VlcMovieStim.replay([autoPlay])</code>	Replay the movie from the beginning.
<code>VlcMovieStim.volume</code>	Audio track volume (<i>int</i> or <i>float</i>).
<code>VlcMovieStim.setVolume(volume)</code>	Set the audio track volume.
<code>VlcMovieStim.getVolume()</code>	Returns the current movie audio volume.
<code>VlcMovieStim.increaseVolume([amount])</code>	Increase the volume.

continues on next page

Table 9.30 – continued from previous page

<code>VlcMovieStim.decreaseVolume([amount])</code>	Decrease the volume.
<code>VlcMovieStim.frameIndex</code>	Current frame index being displayed (<i>int</i>).
<code>VlcMovieStim.getCurrentFrameNumber()</code>	Get the current movie frame number (<i>int</i>), same as <i>frameIndex</i> .
<code>VlcMovieStim.duration</code>	Duration of the loaded video in seconds (<i>float</i>).
<code>VlcMovieStim.loopCount</code>	Number of loops completed since playback started (<i>int</i>).
<code>VlcMovieStim.fps</code>	Movie frames per second (<i>float</i>).
<code>VlcMovieStim.getFPS()</code>	Movie frames per second.
<code>VlcMovieStim.frameTime</code>	Current frame time in seconds (<i>float</i>).
<code>VlcMovieStim.getCurrentFrameTime()</code>	Get the time that the movie file specified the current video frame as having.
<code>VlcMovieStim.percentageComplete</code>	Percentage of the video completed (<i>float</i>).
<code>VlcMovieStim.getPercentageComplete()</code>	Provides a value between 0.0 and 100.0, indicating the amount of the movie that has been already played.
<code>VlcMovieStim.videoSize</code>	Size of the video (<i>w, h</i>) in pixels (<i>tuple</i>).
<code>VlcMovieStim.interpolate</code>	Enable linear interpolation (<code>`bool`</code>).
<code>VlcMovieStim.setFlipHoriz([newVal, log])</code>	If set to True then the movie will be flipped horizontally (left-to-right).
<code>VlcMovieStim.setFlipVert([newVal, log])</code>	If set to True then the movie will be flipped vertically (top-to-bottom).
<code>VlcMovieStim.filename</code>	File name for the loaded video (<i>str</i>).
<code>VlcMovieStim.autoStart</code>	Start playback when <code>.draw()</code> is called (<i>bool</i>).

Details

```
class psychopy.visual.VlcMovieStim(win, filename="", units='pix', size=None, pos=0.0, 0.0,
    ori=0.0, flipVert=False, flipHoriz=False, color=1.0, 1.0,
    1.0, colorSpace='rgb', opacity=1.0, volume=1.0, name="",
    loop=False, autoLog=True, depth=0.0, noAudio=False,
    interpolate=True, autoStart=True)
```

A stimulus class for playing movies in various formats (mpeg, avi, etc...) in PsychoPy using the VLC media player as a decoder.

This movie class is very efficient and better suited for playing high-resolution videos (720p+) than the other movie classes. However, audio is only played using the default output device. This may be adequate for most applications where the user is not concerned about precision audio onset times.

The VLC media player (<https://www.videolan.org/>) must be installed on the machine running PsychoPy to use this class. Make certain that the version of VLC installed matches the architecture of the Python interpreter hosting PsychoPy.

Parameters

- **win** (*Window*) – Window the video is being drawn to.
- **filename** (*str*) – Name of the file or stream URL to play. If an empty string, no file will be loaded on initialization but can be set later.
- **units** (*str*) – Units to use when sizing the video frame on the window, affects how *size* is interpreted.
- **size** (*ArrayLike* or *None*) – Size of the video frame on the window in *units*. If *None*, the native size of the video will be used.
- **flipVert** (*bool*) – If *True* then the movie will be top-bottom flipped.

- **flipHoriz** (*bool*) – If *True* then the movie will be right-left flipped.
- **volume** (*int or float*) – If specifying an *int* the nominal level is 100, and 0 is silence. If a *float*, values between 0 and 1 may be used.
- **loop** (*bool*) – Whether to start the movie over from the beginning if draw is called and the movie is done. Default is `False`.
- **autoStart** (*bool*) – Automatically begin playback of the video when *flip()* is called.

Notes

- You may see error messages in your log output from VLC (e.g., *get_buffer() failed, no frame!*, etc.) after shutting down. These errors originate from the decoder and can be safely ignored.

`_calcPosRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`.

`_calcSizeRendered()`

DEPRECATED in 1.80.00. This functionality is now handled by `_updateVertices()` and `verticesPix`.

`_closeMedia()`

Internal method to release the presently loaded stream (if any).

`_createVLCInstance()`

Internal method to create a new VLC instance.

Raises an error if an instance is already spawned and hasn't been released.

`_drawRectangle()`

Draw the frame to the window. This is called by the *draw()* method.

`_freeBuffers()`

Free texture and pixel buffers. Call this when tearing down this class or if a movie is stopped.

`_getPolyAsRendered()`

DEPRECATED. Return a list of vertices as rendered.

`_onEos()`

Internal method called when the decoder encounters the end of the stream.

`_openMedia(uri=None)`

Internal method that opens a new stream using *filename*. This will close the previous stream. Raises an error if a VLC instance is not available.

`_pixelTransfer()`

Internal method which maps the pixel buffer for the video texture to client memory, allowing for VLC to directly draw a video frame to it.

This method is not thread-safe and should never be called without the pixel lock semaphore being first set by VLC.

`_releaseVLCInstance()`

Internal method to release a VLC instance. Calling this implicitly stops and releases any stream presently loaded and playing.

`_selectWindow(win)`

Switch drawing to the specified window. Calls the window's `_setCurrent()` method which handles the switch.

`_set(attr, val, op="", log=None)`

DEPRECATED since 1.80.04 + 1. Use `setAttribute()` and `val2array()` instead.

`_setupTextureBuffers ()`

Setup texture buffers which hold frame data. This creates a 2D RGB texture and pixel buffer. The pixel buffer serves as the store for texture color data. Each frame, the pixel buffer memory is mapped and frame data is copied over to the GPU from the decoder.

This is called every time a video file is loaded. The `_freeBuffers` method is called in this routine prior to creating new buffers, so it's safe to call this right after loading a new movie without having to `_freeBuffers` first.

`_updateList ()`

The user shouldn't need this method since it gets called after every call to `.set()` Chooses between using and not using shaders each call.

`_updateVertices ()`

Sets `Stim.verticesPix` and `._borderPix` from `pos`, `size`, `ori`, `flipVert`, `flipHoriz`

property `anchor`

`autoDraw`

Determines whether the stimulus should be automatically drawn on every frame flip.

Value should be: *True* or *False*. You do NOT need to set this on every frame flip!

`autoLog`

Whether every change in this stimulus should be auto logged.

Value should be: *True* or *False*. Set to *False* if your stimulus is updating frequently (e.g. updating its position every frame) and you want to avoid swamping the log file with messages that aren't likely to be useful.

property `autoStart`

Start playback when `.draw()` is called (*bool*).

`contains (x, y=None, units=None)`

Returns True if a point `x,y` is inside the stimulus' border.

Can accept variety of input options:

- two separate args, `x` and `y`
- one arg (list, tuple or array) containing two vals (`x,y`)
- **an object with a `getPos()` method that returns `x,y`, such** as a *Mouse*.

Returns *True* if the point is within the area defined either by its *border* attribute (if one defined), or its *vertices* attribute if there is no `.border`. This method handles complex shapes, including concavities and self-crossings.

Note that, if your stimulus uses a mask (such as a Gaussian) then this is not accounted for by the *contains* method; the extent of the stimulus is determined purely by the `size`, `position (pos)`, and `orientation (ori)` settings (and by the `vertices` for shape stimuli).

See Coder demos: `shapeContains.py` See Coder demos: `shapeContains.py`

`decreaseVolume (amount=10)`

Decrease the volume.

Parameters `amount (int)` – Decrease the volume by this amount (percent). This gets subtracted from the present volume level. If the value of *amount* and the current volume is outside the valid range of 0 to 100, the value will be clipped. The default value is 10 (or 10% decrease).

Returns Volume after changed.

Return type `int`

See also:

`getVolume()`, `setVolume()`, `increaseVolume()`

Examples

Adjust the volume of the current video using key presses:

```
# assume `mov` is an instance of this class defined previously
for key in event.getKeys():
    if key == 'minus':
        mov.decreaseVolume()
    elif key == 'equals':
        mov.increaseVolume()
```

depth

DEPRECATED, depth is now controlled simply by drawing order.

draw (*win=None*)

Draw the current frame to a particular *Window* (or to the default win for this object if not specified).

The current position in the movie will be determined automatically. This method should be called on every frame that the movie is meant to appear.

Parameters *win* (*Window* or *None*) – Window the video is being drawn to. If *None*, the window specified at initialization will be used instead.

Returns *True* if the frame was updated this draw call.

Return type `bool`

property duration

Duration of the loaded video in seconds (*float*). Not valid unless the video has been started.

fastForward (*seconds=5*)

Fast-forward the video.

Parameters *seconds* (*float*) – Time in seconds to fast forward from the current position. Default is 5 seconds.

Returns Timestamp at new position after fast forwarding the video.

Return type `float`

property filename

File name for the loaded video (*str*).

property flip

1x2 array for flipping vertices along each axis; set as *True* to flip or *False* to not flip. If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

property flipVert

property fps

Movie frames per second (*float*).

property frameIndex

Current frame index being displayed (*int*).

property frameTexture

Texture ID for the current video frame (*GLuint*). You can use this as a video texture. However, you must periodically call *updateTexture* to keep this up to date.

property frameTime

Current frame time in seconds (*float*).

getCurrentFrameNumber ()

Get the current movie frame number (*int*), same as *frameIndex*.

getCurrentFrameTime ()

Get the time that the movie file specified the current video frame as having.

Returns Current video time in seconds.

Return type *float*

getFPS ()

Movie frames per second.

Returns Nominal number of frames to be displayed per second.

Return type *float*

getPercentageComplete ()

Provides a value between 0.0 and 100.0, indicating the amount of the movie that has been already played.

getVolume ()

Returns the current movie audio volume.

Returns Volume level, 0 is no audio, 100 is max audio volume.

Return type *int*

property height

increaseVolume (amount=10)

Increase the volume.

Parameters **amount** (*int*) – Increase the volume by this amount (percent). This gets added to the present volume level. If the value of *amount* and the current volume is outside the valid range of 0 to 100, the value will be clipped. The default value is 10 (or 10% increase).

Returns Volume after changed.

Return type *int*

See also:

getVolume (), *setVolume ()*, *decreaseVolume ()*

Examples

Adjust the volume of the current video using key presses:

```
# assume `mov` is an instance of this class defined previously
for key in event.getKeys():
    if key == 'minus':
        mov.decreaseVolume()
    elif key == 'equals':
        mov.increaseVolume()
```

property interpolate

Enable linear interpolation (`bool`).

If *True* linear filtering will be applied to the video making the image less pixelated if scaled. You may leave this off if the native size of the video is used.

property isFinished

True if the video is finished (*bool*).

property isNotStarted

True if the video has not be started yet (*bool*). This status is given after a video is loaded and play has yet to be called.

property isPaused

True if the video is presently paused (*bool*).

property isPlaying

True if the video is presently playing (*bool*).

property isStopped

True if the video is stopped (*bool*).

loadMovie (*filename*, *log=True*)

Load a movie from file

Parameters

- **filename** (*str*) – The name of the file or URL, including path if necessary.
- **log** (*bool*) – Log this event.

Notes

- Due to VLC oddness, *.duration* is not correct until the movie starts playing.

property loopCount

Number of loops completed since playback started (*int*). This value is reset when either *stop* or *loadMovie* is called.

name

The name (*str*) of the object to be using during logged messages about this stim. If you have multiple stimuli in your experiment this really helps to make sense of log files!

If name = None your stimulus will be called “unnamed <type>”, e.g. `visual.TextStim(win)` will be called “unnamed TextStim” in the logs.

property opacity

Determines how visible the stimulus is relative to background.

The value should be a single float ranging 1.0 (opaque) to 0.0 (transparent). *Operations* are supported. Precisely how this is used depends on the *Blend Mode*.

ori

The orientation of the stimulus (in degrees).

Should be a single value (*scalar*). *Operations* are supported.

Orientation convention is like a clock: 0 is vertical, and positive values rotate clockwise. Beyond 360 and below zero values wrap appropriately.

overlaps (*polygon*)

Returns *True* if this stimulus intersects another one.

If *polygon* is another stimulus instance, then the vertices and location of that stimulus will be used as the polygon. Overlap detection is typically very good, but it can fail with very pointy shapes in a crossed-swords configuration.

Note that, if your stimulus uses a mask (such as a Gaussian blob) then this is not accounted for by the *overlaps* method; the extent of the stimulus is determined purely by the size, pos, and orientation settings (and by the vertices for shape stimuli).

See coder demo, `shapeContains.py`

pause (*log=True*)

Pause the current point in the movie.

Parameters `log` (*bool*) – Log the pause event.

property percentageComplete

Percentage of the video completed (*float*).

play (*log=True*)

Start or continue a paused movie from current position.

Parameters `log` (*bool*) – Log the play event.

Returns Frame index playback started at. Should always be 0 if starting at the beginning of the video. Returns *None* if the player has not been initialized.

Return type `int` or *None*

property pos

The position of the center of the stimulus in the stimulus *units* *value* should be an *x,y-pair*. *Operations* are also supported.

Example:

```
stim.pos = (0.5, 0) # Set slightly to the right of center
stim.pos += (0.5, -1) # Increment pos rightwards and upwards.
    Is now (1.0, -1.0)
stim.pos *= 0.2 # Move stim towards the center.
    Is now (0.2, -0.2)
```

Tip: If you need the position of stim in pixels, you can obtain it like this:

```
from psychopy.tools.monitorunittools import posToPix
posPix = posToPix(stim)
```

replay (*autoPlay=True*)

Replay the movie from the beginning.

Parameters `autoPlay` (*bool*) – Start playback immediately. If *False*, you must call *play()* afterwards to initiate playback.

Notes

- This tears down the current VLC instance and creates a new one. Similar to calling `stop()` and `loadMovie()`. Use `seek(0.0)` if you would like to restart the movie without reloading.

rewind (*seconds=5*)

Rewind the video.

Parameters **seconds** (*float*) – Time in seconds to rewind from the current position. Default is 5 seconds.

Returns Timestamp after rewinding the video.

Return type *float*

seek (*timestamp, log=True*)

Seek to a particular timestamp in the movie.

Parameters

- **timestamp** (*float*) – Time in seconds.
- **log** (*bool*) – Log the seek event.

setAnchor (*value, log=None*)

setAutoDraw (*val, log=None*)

Add or remove a stimulus from the list of stimuli that will be automatically drawn on each flip

Parameters

- **val: True/False** True to add the stimulus to the draw list, False to remove it

setAutoLog (*value=True, log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setDepth (*newDepth, operation="", log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setFlipHoriz (*newVal=True, log=True*)

If set to True then the movie will be flipped horizontally (left-to-right). Note that this is relative to the original, not relative to the current state.

setFlipVert (*newVal=True, log=True*)

If set to True then the movie will be flipped vertically (top-to-bottom). Note that this is relative to the original, not relative to the current state.

setMovie (*filename, log=True*)

See `~MovieStim.loadMovie` (the functions are identical).

This form is provided for syntactic consistency with other visual stimuli.

setOpacity (*newOpacity, operation="", log=None*)

Hard setter for opacity, allows the suppression of log messages and calls the update method

setOri (*newOri, operation="", log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message

setPos (*newPos, operation="", log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setSize (*newSize*, *operation=""*, *units=None*, *log=None*)

Usually you can use 'stim.attribute = value' syntax instead, but use this method if you need to suppress the log message

setVolume (*volume*)

Set the audio track volume.

Parameters volume (*int or float*) – Volume level to set. 0 = mute, 100 = 0 dB. float values between 0.0 and 1.0 are also accepted, and scaled to an int between 0 and 100.

property size

The size (width, height) of the stimulus in the stimulus *units*

Value should be *x,y-pair*, *scalar* (applies to both dimensions) or None (resets to default). *Operations* are supported.

Sizes can be negative (causing a mirror-image reversal) and can extend beyond the window.

Example:

```
stim.size = 0.8 # Set size to (xsize, ysize) = (0.8, 0.8)
print(stim.size) # Outputs array([0.8, 0.8])
stim.size += (0.5, -0.5) # make wider and flatter: (1.3, 0.3)
```

Tip: if you can see the actual pixel range this corresponds to by looking at *stim._sizeRendered*

stop (*log=True*)

Stop the current point in the movie (sound will stop, current frame will not advance). Once stopped the movie cannot be restarted - it must be loaded again.

Use *pause()* instead if you may need to restart the movie.

Parameters log (*bool*) – Log the stop event.

property units

updateOpacity ()

Placeholder method to update colours when set externally, for example updating the *palette* attribute of a textbox.

updateTexture ()

Update the video texture buffer to the most recent video frame.

property vertices

property verticesPix

This determines the coordinates of the vertices for the current stimulus in pixels, accounting for size, ori, pos and units

property videoSize

Size of the video (*w, h*) in pixels (*tuple*). Returns (*0, 0*) if no video is loaded.

property volume

Audio track volume (*int or float*). See *setVolume* for more information about valid values.

property width

property win

The *Window* object in which the stimulus will be rendered by default. (required)

Example, drawing same stimulus in two different windows and display simultaneously. Assuming that you have two windows and a stimulus (*win1*, *win2* and *stim*):

```
stim.win = win1 # stimulus will be drawn in win1
stim.draw() # stimulus is now drawn to win1
stim.win = win2 # stimulus will be drawn in win2
stim.draw() # it is now drawn in win2
win1.flip(waitBlanking=False) # do not wait for next
# monitor update
win2.flip() # wait for vertical blanking.
```

Note that this just changes **default** window for stimulus.

You could also specify window-to-draw-to when drawing:

```
stim.draw(win1)
stim.draw(win2)
```

9.3.38 psychopy.visual.VisualSystemHD

Classes for using NordicNeuralLab’s VisualSystemHD in-scanner display for presenting visual stimuli. Support is preliminary so users must empirically verify whether the default settings for barrel distortion and FOV are correct. Support may be good enough at this point for studies that do not require precise stereoscopy or stimulus sizes.

Overview

<code>VisualSystemHD([monoscopic, diopters, ...])</code>	Class provides support for NordicNeuralLab’s VisualSystemHD(tm) fMRI display hardware.
<code>VisualSystemHD.monoscopic</code>	<i>True</i> if using monoscopic mode.
<code>VisualSystemHD.lensCorrection</code>	<i>True</i> if using lens correction.
<code>VisualSystemHD.distCoef</code>	Distortion coefficient (<i>float</i>).
<code>VisualSystemHD.diopters</code>	Diopters value of the current eye buffer.
<code>VisualSystemHD.setDiopters(diopters[, eye])</code>	Set the diopters for a given eye.
<code>VisualSystemHD.eyeOffset</code>	Eye offset for the current buffer in centimeters used for stereoscopic rendering.
<code>VisualSystemHD.setEyeOffset(dist[, eye])</code>	Set the eye offset in centimeters.
<code>VisualSystemHD.setBuffer(buffer[, clear])</code>	Set the eye buffer to draw to.
<code>VisualSystemHD.setPerspectiveView(...)</code>	Set the projection and view matrix to render with perspective.

Details

```
class psychopy.visual.nnlvs.VisualSystemHD (monoscopic=False, diopters=- 1, - 1, lensCorrection=True, distCoef=None, directDraw=False, model='vshd', *args, **kwargs)
```

Class provides support for NordicNeuralLab’s VisualSystemHD(tm) fMRI display hardware.

Use this class in-place of the *Window* class for use with the VSHD hardware. Ensure that the VSHD headset display output is configured in extended desktop mode (eg. nVidia Surround). Extended desktops are only supported on Windows and Linux systems.

The VSHD is capable of both 2D and stereoscopic 3D rendering. You can select which eye to draw to by calling *setBuffer*, much like how stereoscopic rendering is implemented in the base *Window* class.

Notes

- This class handles drawing differently than the default window class, as a result, stimuli *autoDraw* is not supported.
- Edges of the warped image may appear jagged. To correct this, create a window using *multiSample=True* and *numSamples > 1* to smooth out these artifacts.

Examples

Here is a basic example of 2D rendering using the VisualSystemHD(tm). This is the binocular version of the dynamic ‘plaid.py’ demo:

```
from psychopy import visual, core, event

# Create a visual window
win = visual.VisualSystemHD(fullscr=True, screen=1)

# Initialize some stimuli, note contrast, opacity, ori
grating1 = visual.GratingStim(win, mask="circle", color='white',
    contrast=0.5, size=(1.0, 1.0), sf=(4, 0), ori = 45, autoLog=False)
grating2 = visual.GratingStim(win, mask="circle", color='white',
    opacity=0.5, size=(1.0, 1.0), sf=(4, 0), ori = -45, autoLog=False,
    pos=(0.1, 0.1))

trialClock = core.Clock()
t = 0
while not event.getKeys() and t < 20:
    t = trialClock.getTime()

    for eye in ('left', 'right'):
        win.setBuffer(eye) # change the buffer
        grating1.phase = 1 * t # drift at 1Hz
        grating1.draw() # redraw it
        grating2.phase = 2 * t # drift at 2Hz
        grating2.draw() # redraw it

    win.flip()

win.close()
core.quit()
```

As you can see above, there are few changes needed to convert an existing 2D experiment to run on the VSHD. For 3D rendering with perspective, you need set *eyeOffset* and apply the projection by calling *setPerspectiveView*. (other projection modes are not implemented or supported right now):

```
from psychopy import visual, core, event

# Create a visual window
win = visual.VisualSystemHD(fullscr=True, screen=1,
    multiSample=True, nSamples=8)

# text to display
instr = visual.TextStim(win, text="Any key to quit", pos=(0, -.7))

# create scene light at the pivot point
win.lights = [
```

(continues on next page)

(continued from previous page)

```

        visual.LightSource(win, pos=(0.4, 4.0, -2.0), lightType='point',
                           diffuseColor=(0, 0, 0), specularColor=(1, 1, 1))
    ]
    win.ambientLight = (0.2, 0.2, 0.2)

    # Initialize some stimuli, note contrast, opacity, ori
    ball = visual.SphereStim(win, radius=0.1, pos=(0, 0, -2), color='green',
                              useShaders=False)

    iod = 6.2 # interocular separation in CM
    win.setEyeOffset(-iod / 2.0, 'left')
    win.setEyeOffset(iod / 2.0, 'right')

    trialClock = core.Clock()
    t = 0
    while not event.getKeys() and t < 20:
        t = trialClock.getTime()

        for eye in ('left', 'right'):
            win.setBuffer(eye) # change the buffer

            # setup drawing with perspective
            win.setPerspectiveView()

            win.useLights = True # switch on lights
            ball.draw() # draw the ball
            # shut the lights, needed to prevent light color from affecting
            # 2D stim
            win.useLights = False

            # reset transform to draw text correctly
            win.resetEyeTransform()

            instr.draw()

        win.flip()

    win.close()
    core.quit()

```

Parameters

- **monoscopic** (*bool*) – Use monoscopic rendering. If *True*, the same image will be drawn to both eye buffers. You will not need to call *setBuffer*. It is not possible to set monoscopic mode after the window is created. It is recommended that you use monoscopic mode if you intend to display only 2D stimuli about the center of the display as it uses a less memory intensive rendering pipeline.
- **diopeters** (*tuple or list*) – Initial diopter values for the left and right eye. Default is *(-1, -1)*, values must be integers.
- **lensCorrection** (*bool*) – Apply lens correction (barrel distortion) to the output. The amount of distortion applied can be specified using *distCoef*. If *False*, no distortion will be applied to the output and the entire display will be used. Not applying correction will result in pincushion distortion which produces a non-rectilinear output.
- **distCoef** (*float*) – Distortion coefficient for barrel distortion. If *None*, the recom-

mended value will be used for the model of display. You can adjust the value to fine-tune the barrel distortion.

- **directDraw** (*bool*) – Direct drawing mode. Stimuli are drawn directly to the back buffer instead of creating separate buffer. This saves video memory but does not permit barrel distortion or monoscopic rendering. If *False*, drawing is done with two FBOs containing each eye’s image.
- **hwModel** (*str*) – Model of the VisualSystemHD in use. Used to set viewing parameters accordingly. Default is ‘vshd’. Cannot be changed after starting the application.

__assignFlipTime (*obj, attrib*)

Helper function to assign the time of last flip to the obj.attrib

Parameters

- **obj** (*dict or object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of obj to assign the flip time to.

__blitEyeBuffer (*eye*)

Warp and blit to the appropriate eye buffer.

Parameters eye (*str*) – Eye buffer being used.

__checkMatchingSizes (*requested, actual*)

Checks whether the requested and actual screen sizes differ. If not then a warning is output and the window size is set to actual

__cleanEditables ()

Make sure there are no dead refs in the editables list

__endOffFlip (*clearBuffer*)

Override end of flip with custom color channel masking if required.

__getFrame (*rect=None, buffer='front'*)

Return the current Window as an image.

__getRegionOfFrame (*rect=- 1, 1, 1, - 1, buffer='front', power2=False, squarePower2=False*)

Deprecated function, here for historical reasons. You may now use `:py:attr: `~Window._getFrame()` and specify a rect to get a sub-region, just as used here.

power2 can be useful with older OpenGL versions to avoid interpolation in PatchStim. If power2 or squarePower2, it will expand rect dimensions up to next power of two. squarePower2 uses the max dimensions. You need to check what your hardware & OpenGL supports, and call `__getRegionOfFrame()` as appropriate.

__getWarpExtents (*eye*)

Get the horizontal and vertical extents of the barrel distortion in normalized device coordinates. This is used to determine the FOV along each axis after barrel distortion.

Parameters eye (*str*) – Eye to compute the extents for.

Returns 2d array of coordinates [+X, -X, +Y, -Y] of the extents of the barrel distortion.

Return type ndarray

__renderFBO ()

Perform a warp operation.

(in this case a copy operation without any warping)

`_setCurrent()`

Make this window's OpenGL context current.

If called on a window whose context is current, the function will return immediately. This reduces the number of redundant calls if no context switch is required. If `useFBO=True`, the framebuffer is bound after the context switch.

`_setupEyeBuffers()`

Setup additional buffers for rendering content to each eye.

`_setupGL()`

Setup OpenGL state for this window.

`_setupGamma(gammaVal)`

A private method to work out how to handle gamma for this Window given that the user might have specified an explicit value, or maybe gave a Monitor.

`_setupLensCorrection()`

Setup the VAOs needed for lens correction.

`_startOfFlip()`

Custom `_startOfFlip` for HMD rendering. This finalizes the HMD texture before diverting drawing operations back to the on-screen window. This allows `flip` to swap the on-screen and HMD buffers when called. This function always returns *True*.

Returns

Return type True

`addEditable(editable)`

Adds an editable element to the screen (something to which characters can be sent with meaning from the keyboard).

The current editable object receiving chars is `Window.currentEditable`

Parameters `editable` –

Returns

property `ambientLight`

Ambient light color for the scene [r, g, b, a]. Values range from 0.0 to 1.0. Only applicable if `useLights` is *True*.

Examples

Setting the ambient light color:

```
win.ambientLight = [0.5, 0.5, 0.5]

# don't do this!!!
win.ambientLight[0] = 0.5
win.ambientLight[1] = 0.5
win.ambientLight[2] = 0.5
```

`applyEyeTransform(clearDepth=True)`

Apply the current view and projection matrices.

Matrices specified by attributes `viewMatrix` and `projectionMatrix` are applied using 'immediate mode' OpenGL functions. Subsequent drawing operations will be affected until `flip()` is called.

All transformations in `GL_PROJECTION` and `GL_MODELVIEW` matrix stacks will be cleared (set to identity) prior to applying.

Parameters `clearDepth` (*bool*) – Clear the depth buffer. This may be required prior to rendering 3D objects.

Examples

Using a custom view and projection matrix:

```
# Must be called every frame since these values are reset after
# `flip()` is called!
win.viewMatrix = viewtools.lookAt( ... )
win.projectionMatrix = viewtools.perspectiveProjectionMatrix( ... )
win.applyEyeTransform()
# draw 3D objects here ...
```

property aspect

Aspect ratio of the current viewport (width / height).

blendMode

Blend mode to use.

callOnFlip (*function, *args, **kwargs*)

Call a function immediately after the next `flip()` command.

The first argument should be the function to call, the following args should be used exactly as you would for your normal call to the function (can use ordered arguments or keyword arguments as normal).

e.g. If you have a function that you would normally call like this:

```
pingMyDevice(portToPing, channel=2, level=0)
```

then you could call `callOnFlip()` to have the function call synchronized with the frame flip like this:

```
win.callOnFlip(pingMyDevice, portToPing, channel=2, level=0)
```

clearBuffer (*color=True, depth=False, stencil=False*)

Clear the present buffer (to which you are currently drawing) without flipping the window.

Useful if you want to generate movie sequences from the back buffer without actually taking the time to flip the window.

Set *color* prior to clearing to set the color to clear the color buffer to. By default, the depth buffer is cleared to a value of 1.0.

Parameters

- **color** (*bool*) – Buffers to clear.
- **depth** (*bool*) – Buffers to clear.
- **stencil** (*bool*) – Buffers to clear.

Examples

Clear the color buffer to a specified color:

```
win.color = (1, 0, 0)
win.clearBuffer(color=True)
```

Clear only the depth buffer, *depthMask* must be *True* or else this will have no effect. Depth mask is usually *True* by default, but may change:

```
win.depthMask = True
win.clearBuffer(color=False, depth=True, stencil=False)
```

close()

Close the window (and reset the Bits++ if necess).

property color

Set the color of the window.

This command sets the color that the blank screen will have on the next clear operation. As a result it effectively takes TWO `flip()` operations to become visible (the first uses the color to create the new screen, the second presents that screen to the viewer). For this reason, if you want to changed background color of the window “on the fly”, it might be a better idea to draw a `Rect` that fills the whole window with the desired `Rect.fillColor` attribute. That’ll show up on first flip.

See other stimuli (e.g. `GratingStim.color`) for more info on the color attribute which essentially works the same on all PsychoPy stimuli.

See [Color spaces](#) for further information about the ways to specify colors and their various implications.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property contentScaleFactor

Scaling factor (*float*) to use when drawing to the backbuffer to convert framebuffer to client coordinates.

See also:

[*getContentScaleFactor*](#)

property convergeOffset

Convergence offset from monitor in centimeters.

This value corresponds to the offset from screen plane to set the convergence plane (or point for *toe-in* projections). Positive offsets move the plane farther away from the viewer, while negative offsets nearer. This value is used by *setPerspectiveView* and should be set before calling it to take effect.

Notes

- This value is only applicable for *setToeIn* and *setOffAxisView*.

coordToRay (*screenXY*)

Convert a screen coordinate to a direction vector.

Takes a screen/window coordinate and computes a vector which projects a ray from the viewpoint through it (line-of-sight). Any 3D point touching the ray will appear at the screen coordinate.

Uses the current *viewport* and *projectionMatrix* to calculate the vector. The vector is in eye-space, where the origin of the scene is centered at the viewpoint and the forward direction aligned with the -Z axis. A ray of (0, 0, -1) results from a point at the very center of the screen assuming symmetric frustums.

Note that if you are using a flipped/mirrored view, you must invert your supplied screen coordinates (*screenXY*) prior to passing them to this function.

Parameters **screenXY** (*array_like*) – X, Y screen coordinate. Must be in units of the window.

Returns Normalized direction vector [x, y, z].

Return type ndarray

Examples

Getting the direction vector between the mouse cursor and the eye:

```
mx, my = mouse.getPos()
dir = win.coordToRay((mx, my))
```

Set the position of a 3D stimulus object using the mouse, constrained to a plane. The object origin will always be at the screen coordinate of the mouse cursor:

```
# the eye position in the scene is defined by a rigid body pose
win.viewMatrix = camera.getViewMatrix()
win.applyEyeTransform()

# get the mouse location and calculate the intercept
mx, my = mouse.getPos()
ray = win.coordToRay([mx, my])
result = intersectRayPlane( # from mathtools
    orig=camera.pos,
    dir=camera.transformNormal(ray),
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))

# if result is `None`, there is no intercept
if result is not None:
    pos, dist = result
    objModel.thePose.pos = pos
else:
    objModel.thePose.pos = (0, 0, -10) # plane origin
```

If you don't define the position of the viewer with a *RigidBodyPose*, you can obtain the appropriate eye position and rotate the ray by doing the following:

```
pos = numpy.linalg.inv(win.viewMatrix)[:3, 3]
ray = win.coordToRay([mx, my]).dot(win.viewMatrix[:3, :3])
# then ...
result = intersectRayPlane(
    orig=pos,
    dir=ray,
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))
```

property cullFace

True if face culling is enabled.

property cullFaceMode

Face culling mode, either *back*, *front* or *both*.

property currentEditable

The editable (Text?) object that currently has key focus

property depthFunc

Depth test comparison function for rendering.

property depthMask

True if depth masking is enabled. Writing to the depth buffer will be disabled.

property depthTest

True if depth testing is enabled.

property diopters

Diopters value of the current eye buffer.

classmethod dispatchAllWindowsEvents()

Dispatches events for all pyglet windows. Used by iohub 2.0 psychopy kb event integration.

property distCoef

Distortion coefficient (*float*).

property draw3d

True if 3D drawing is enabled on this window.

property eyeOffset

Eye offset for the current buffer in centimeters used for stereoscopic rendering. This works differently than the main window class as it sets the offset for the current buffer. The offset is saved and automatically restored when the buffer is selected.

property farClip

Distance to the far clipping plane in meters.

flip (*clearBuffer=True*)

Flip the front and back buffers after drawing everything for your frame. (This replaces the `update()` method, better reflecting what is happening underneath).

Parameters `clearBuffer` (*bool*, *optional*) – Clear the draw buffer after flipping. Default is *True*.

Returns Wall-clock time in seconds the flip completed. Returns *None* if `waitBlanking` is *False*.

Return type `float` or `None`

Notes

- The time returned when `waitBlanking` is `True` corresponds to when the graphics driver releases the draw buffer to accept draw commands again. This time is usually close to the vertical sync signal of the display.

Examples

Results in a clear screen after flipping:

```
win.flip(clearBuffer=True)
```

The screen is not cleared (so represent the previous screen):

```
win.flip(clearBuffer=False)
```

`fps()`

Report the frames per second since the last call to this function (or since the window was created if this is first call)

property `frameBufferSize`

Size of the framebuffer in pixels (w, h).

property `frontFace`

Face winding order to define front, either *ccw* or *cw*.

property `fullscr`

Set whether fullscreen mode is *True* or *False* (not all backends can toggle an open window).

property `gamma`

Set the monitor gamma for linearization.

Warning: Don't use this if using a Bits++ or Bits#, as it overrides monitor settings.

property `gammaRamp`

Sets the hardware CLUT using a specified 3xN array of floats ranging between 0.0 and 1.0.

Array must have a number of rows equal to $2^{\max(\text{bpc})}$.

property `getActualFrameRate` (*nIdentical=10, nMaxFrames=100, nWarmUpFrames=10, threshold=1*)

Measures the actual frames-per-second (FPS) for the screen.

This is done by waiting (for a max of *nMaxFrames*) until *nIdentical* frames in a row have identical frame times (std dev below *threshold* ms).

Parameters

- **nIdentical** (*int, optional*) – The number of consecutive frames that will be evaluated. Higher → greater precision. Lower → faster.
- **nMaxFrames** (*int, optional*) – The maximum number of frames to wait for a matching set of *nIdentical*.
- **nWarmUpFrames** (*int, optional*) – The number of frames to display before starting the test (this is in place to allow the system to settle after opening the *Window* for the first time).

- **threshold** (*int or float, optional*) – The threshold for the std deviation (in ms) before the set are considered a match.

Returns Frame rate (FPS) in seconds. If there is no such sequence of identical frames a warning is logged and *None* will be returned.

Return type *float* or *None*

getContentScaleFactor ()

Get the scaling factor required for scaling correctly on high-DPI displays.

If the returned value is 1.0, no scaling needs to be applied to objects drawn on the backbuffer. A value >1.0 indicates that the backbuffer is larger than the reported client area, requiring points to be scaled to maintain constant size across similarly sized displays. In other words, the scaling required to convert framebuffer to client coordinates.

Returns Scaling factor to be applied along both horizontal and vertical dimensions.

Return type *float*

Examples

Get the size of the client area:

```
clientSize = win.frameBufferSize / win.getContentScaleFactor()
```

Get the framebuffer size from the client size:

```
frameBufferSize = win.clientSize * win.getContentScaleFactor()
```

Convert client (window) to framebuffer pixel coordinates (eg., a mouse coordinate, vertices, etc.):

```
# `mousePosXY` is an array ...
frameBufferXY = mousePosXY * win.getContentScaleFactor()
# you can also use the attribute ...
frameBufferXY = mousePosXY * win.contentScaleFactor
```

Notes

- This value is only valid after the window has been fully realized.

getFutureFlipTime (*targetTime=0, clock=None*)

The expected time of the next screen refresh. This is currently calculated as `win._lastFrameTime + refreshInterval`

Parameters

- **targetTime** (*float*) – The delay *from now* for which you want the flip time. 0 will give the because that the earliest we can achieve. 0.15 will give the schedule flip time that gets as close to 150 ms as possible
- **clock** (*None, 'ptb', 'now' or any Clock object*) – If True then the time returned is compatible with `ptb.GetSecs()`
- **verbose** (*bool*) – Set to True to view the calculations along the way

getMovieFrame (*buffer='front'*)

Capture the current Window as an image.

Saves to stack for `saveMovieFrames()`. As of v1.81.00 this also returns the frame as a PIL image

This can be done at any time (usually after a `flip()` command).

Frames are stored in memory until a `saveMovieFrames()` command is issued. You can issue `getMovieFrame()` as often as you like and then save them all in one go when finished.

The back buffer will return the frame that hasn't yet been 'flipped' to be visible on screen but has the advantage that the mouse and any other overlapping windows won't get in the way.

The default front buffer is to be called immediately after a `flip()` and gives a complete copy of the screen at the window's coordinates.

Parameters `buffer` (*str*, optional) – Buffer to capture.

Returns Buffer pixel contents as a PIL/Pillow image object.

Return type Image

getMsPerFrame (*nFrames=60*, *showVisual=False*, *msg=""*, *msDelay=0.0*)

Assesses the monitor refresh rate (average, median, SD) under current conditions, over at least 60 frames.

Records time for each refresh (frame) for *n* frames (at least 60), while displaying an optional visual. The visual is just eye-candy to show that something is happening when assessing many frames. You can also give it text to display instead of a visual, e.g., `msg='(testing refresh rate...)'`; setting `msg` implies `showVisual == False`.

To simulate refresh rate under cpu load, you can specify a time to wait within the loop prior to doing the `flip()`. If $0 < \text{msDelay} < 100$, wait for that long in ms.

Returns timing stats (in ms) of:

- average time per frame, for all frames
- standard deviation of all frames
- median, as the average of 12 frame times around the median (~monitor refresh rate)

Author

- 2010 written by Jeremy Gray

property `lensCorrection`

True if using lens correction.

property `lights`

Scene lights.

This is specified as an array of `~psychopy.visual.LightSource` objects. If a single value is given, it will be converted to a *list* before setting. Set `useLights` to *True* before rendering to enable lighting/shading on subsequent objects. If `lights` is *None* or an empty *list*, no lights will be enabled if `useLights=True`, however, the scene ambient light set with `ambientLight` will be still be used.

Examples

Create a directional light source and add it to scene lights:

```
dirLight = gltools.LightSource((0., 1., 0.), lightType='directional')
win.lights = dirLight # `win.lights` will be a list when accessed!
```

Multiple lights can be specified by passing values as a list:

```
myLights = [gltools.LightSource((0., 5., 0.)),
            gltools.LightSource((-2., -2., 0.))
win.lights = myLights
```

logOnFlip (*msg, level, obj=None*)

Send a log message that should be time-stamped at the next `flip()` command.

Parameters

- **msg** (*str*) – The message to be logged.
- **level** (*int*) – The level of importance for the message.
- **obj** (*object, optional*) – The python object that might be associated with this message if desired.

property monoscopic

True if using monoscopic mode.

mouseVisible

Sets the visibility of the mouse cursor.

If Window was initialized with `allowGUI=False` then the mouse is initially set to invisible, otherwise it will initially be visible.

Usage:

```
win.mouseVisible = False
win.mouseVisible = True
```

multiFlip (*flips=1, clearBuffer=True*)

Flip multiple times while maintaining the display constant. Use this method for precise timing.

WARNING: This function should not be used. See the *Notes* section for details.

Parameters

- **flips** (*int, optional*) – The number of monitor frames to flip. Floats will be rounded to integers, and a warning will be emitted. `Window.multiFlip(flips=1)` is equivalent to `Window.flip()`. Defaults to *1*.
- **clearBuffer** (*bool, optional*) – Whether to clear the screen after the last flip. Defaults to *True*.

Notes

- This function can behave unpredictably, and the PsychoPy authors recommend against using it. See <https://github.com/psychoopy/psychoopy/issues/867> for more information.

Examples

Example of using `multiFlip`:

```
# Draws myStim1 to buffer
myStim1.draw()
# Show stimulus for 4 frames (90 ms at 60Hz)
myWin.multiFlip(clearBuffer=False, flips=6)
# Draw myStim2 "on top of" myStim1
# (because buffer was not cleared above)
myStim2.draw()
# Show this for 2 frames (30 ms at 60Hz)
myWin.multiFlip(flips=2)
# Show blank screen for 3 frames (buffer was cleared above)
myWin.multiFlip(flips=3)
```

property nearClip

Distance to the near clipping plane in meters.

nextEditable()

Moves focus of the cursor to the next editable window

onResize (*width, height*)

A default resize event handler.

This default handler updates the GL viewport to cover the entire window and sets the `GL_PROJECTION` matrix to be orthogonal in window space. The bottom-left corner is (0, 0) and the top-right corner is the width and height of the *Window* in pixels.

Override this event handler with your own to create another projection, for example in perspective.

property projectionMatrix

Projection matrix defined as a 4x4 numpy array.

recordFrameIntervals

Record time elapsed per frame.

Provides accurate measures of frame intervals to determine whether frames are being dropped. The intervals are the times between calls to `flip()`. Set to *True* only during the time-critical parts of the script. Set this to *False* while the screen is not being updated, i.e., during any slow, non-frame-time-critical sections of your code, including `inter-trial-intervals`, `event.waitkeys()`, `core.wait()`, or `image.setImage()`.

Examples

Enable frame interval recording, successive frame intervals will be stored:

```
win.recordFrameIntervals = True
```

Frame intervals can be saved by calling the `saveFrameIntervals` method:

```
win.saveFrameIntervals()
```

removeEditable (*editable*)

resetEyeTransform (*clearDepth=True*)

Restore the default projection and view settings to PsychoPy defaults. Call this prior to drawing 2D stimuli objects (i.e. `GratingStim`, `ImageStim`, `Rect`, etc.) if any eye transformations were applied for the stimuli to be drawn correctly.

Parameters **clearDepth** (*bool*) – Clear the depth buffer upon reset. This ensures successive draw commands are not affected by previous data written to the depth buffer. Default is *True*.

Notes

- Calling `flip()` automatically resets the view and projection to defaults. So you don't need to call this unless you are mixing 3D and 2D stimuli.

Examples

Going between 3D and 2D stimuli:

```
# 2D stimuli can be drawn before setting a perspective projection
win.setPerspectiveView()
# draw 3D stimuli here ...
win.resetEyeTransform()
# 2D stimuli can be drawn here again ...
win.flip()
```

resetViewport ()

Reset the viewport to cover the whole framebuffer.

Set the viewport to match the dimensions of the back buffer or framebuffer (if *useFBO=True*). The scissor rectangle is also set to match the dimensions of the viewport.

property `rgb`

saveFrameIntervals (*fileName=None, clear=True*)

Save recorded screen frame intervals to disk, as comma-separated values.

Parameters

- **fileName** (*None* or *str*) – *None* or the filename (including path if necessary) in which to store the data. If *None* then 'lastFrameIntervals.log' will be used.
- **clear** (*bool*) – Clear buffer frames intervals were stored after saving. Default is *True*.

saveMovieFrames (*fileName, codec='libx264', fps=30, clearFrames=True*)

Writes any captured frames to disk.

Will write any format that is understood by PIL (tif, jpg, png, ...)

Parameters

- **filename** (*str*) – Name of file, including path. The extension at the end of the file determines the type of file(s) created. If an image type (e.g. .png) is given, then multiple static frames are created. If it is .gif then an animated GIF image is created (although you will get higher quality GIF by saving PNG files and then combining them in dedicated image manipulation software, such as GIMP). On Windows and Linux .mpeg files can be created if *pymedia* is installed. On macOS .mov files can be created if the pyobjc-frameworks-UIKit is installed. Unfortunately the libs used for movie generation can be flaky and poor quality. As for animated GIFs, better results can be achieved by saving as individual .png frames and then combining them into a movie using software like ffmpeg.
- **codec** (*str, optional*) – The codec to be used by **moviepy** for mp4/mpg/mov files. If *None* then the default will depend on file extension. Can be one of `libx264`, `mpeg4` for mp4/mov files. Can be `rawvideo`, `png` for avi files (not recommended). Can be `libvorbis` for ogv files. Default is `libx264`.
- **fps** (*int, optional*) – The frame rate to be used throughout the movie. **Only for quicktime (.mov) movies..** Default is `30`.
- **clearFrames** (*bool, optional*) – Set this to *False* if you want the frames to be kept for additional calls to `saveMovieFrames`. Default is *True*.

Examples

Writes a series of static frames as `frame001.tif`, `frame002.tif` etc.:

```
myWin.saveMovieFrames('frame.tif')
```

As of PsychoPy 1.84.1 the following are written with **moviepy**:

```
myWin.saveMovieFrames('stimuli.mp4') # codec = 'libx264' or 'mpeg4'
myWin.saveMovieFrames('stimuli.mov')
myWin.saveMovieFrames('stimuli.gif')
```

property scissor

Scissor rectangle (*x, y, w, h*) for the current draw buffer.

Values *x* and *y* define the origin, and *w* and *h* the size of the rectangle in pixels. The scissor operation is only active if *scissorTest=True*.

Usually, the scissor and viewport are set to the same rectangle to prevent drawing operations from *spilling* into other regions of the screen. For instance, calling `clearBuffer` will only clear within the scissor rectangle.

Setting the scissor rectangle but not the viewport will restrict drawing within the defined region (like a rectangular aperture), not changing the positions of stimuli.

property scissorTest

True if scissor testing is enabled.

property screenshot

setBlendMode (*blendMode, log=None*)

Usually you can use `'stim.attribute = value'` syntax instead, but use this method if you need to suppress the log message.

setBuffer (*buffer, clear=True*)

Set the eye buffer to draw to. Subsequent draw calls will be diverted to the specified eye.

Parameters

- **buffer** (*str*) – Eye buffer to draw to. Values can either be ‘left’ or ‘right’.
- **clear** (*bool*) – Clear the buffer prior to drawing.

setColor (*color, colorSpace=None, operation="", log=None*)

Usually you can use `stim.attribute = value` syntax instead, but use this method if you want to set color and colorSpace simultaneously.

See `color` for documentation on colors.

setDiopters (*diopters, eye=None*)

Set the diopters for a given eye.

Parameters

- **diopters** (*int*) – Set diopters for a given eye, ranging between -7 and +5.
- **eye** (*str or None*) – Eye to set, either ‘left’ or ‘right’. If *None*, the currently set buffer will be used.

setEyeOffset (*dist, eye=None*)

Set the eye offset in centimeters.

When set, successive rendering operations will use the new offset.

Parameters

- **dist** (*float or int*) – Lateral offset in centimeters from the nose, usually half the interocular separation. The distance is signed.
- **eye** (*str or None*) – Eye offset to set. Can either be ‘left’, ‘right’ or *None*. If *None*, the offset of the current buffer is used.

setGamma (*gamma, log=None*)

Usually you can use ‘`stim.attribute = value`’ syntax instead, but use this method if you need to suppress the log message.

setMouseEvent (*name='arrow'*)

Change the appearance of the cursor for this window. Cursor types provide contextual hints about how to interact with on-screen objects.

The graphics used ‘standard cursors’ provided by the operating system. They may vary in appearance and hot spot location across platforms. The following names are valid on most platforms:

- `arrow`: Default pointer.
- `ibeam`: Indicates text can be edited.
- `crosshair`: Crosshair with hot-spot at center.
- `hand`: A pointing hand.
- `hresize`: Double arrows pointing horizontally.
- `vresize`: Double arrows pointing vertically.

Parameters name (*str*) – Type of standard cursor to use (see above). Default is `arrow`.

Notes

- On Windows the `crosshair` option is negated with the background color. It will not be visible when placed over 50% grey fields.

setMouseVisible (*visibility, log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setOffAxisView (*applyTransform=True, clearDepth=True*)

Set an off-axis projection.

Create an off-axis projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. If *eyeOffset* is correct and the view distance and screen size is defined in the monitor configuration, the resulting view will approximate *ortho-stereo* viewing.

The convergence plane can be adjusted by setting *convergeOffset*. By default, the convergence plane is set to the screen plane. Any points on the screen plane will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

setPerspectiveView (*applyTransform=True, clearDepth=True*)

Set the projection and view matrix to render with perspective.

Matrices are computed using values specified in the monitor configuration with the scene origin on the screen plane. Calculations assume units are in meters. If *eyeOffset* $\neq 0$, the view will be transformed laterally, however the frustum shape will remain the same.

Note that the values of *projectionMatrix* and *viewMatrix* will be replaced when calling this function.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards if *False*.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

setRGB (*newRGB*)

Deprecated: As of v1.61.00 please use `setColor()` instead

setRecordFrameIntervals (*value=True, log=None*)

Usually you can use ‘stim.attribute = value’ syntax instead, but use this method if you need to suppress the log message.

setScale (*units, font='dummyFont', prevScale=1.0, 1.0*)

DEPRECATED: this method used to be used to switch between units for stimulus drawing but this is now handled by the stimuli themselves and the window should always be left in units of ‘pix’

setToeInView (*applyTransform=True, clearDepth=True*)

Set toe-in projection.

Create a toe-in projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. The value of *convergeOffset* will define the convergence point of the view, which is offset perpendicular to the center of the screen plane. Points falling on a vertical line at the convergence point will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling `applyEyeTransform()` afterwards.
- **clearDepth** (*bool*, *optional*) – Clear the depth buffer.

Notes

- This projection mode is only ‘correct’ if the viewer’s eyes are converged at the convergence point. Due to perspective, this projection introduces vertical disparities which increase in magnitude with eccentricity. Use *setOffAxisView* if you want to display something the viewer can look around the screen comfortably.

setUnits (*value*, *log=True*)

setViewPos (*value*, *log=True*)

property size

Size of the drawable area in pixels (w, h).

property stencilTest

True if stencil testing is enabled.

timeOnFlip (*obj*, *attrib*)

Retrieves the time on the next flip and assigns it to the *attrib* for this *obj*.

Parameters

- **obj** (*dict or object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of *obj* to assign the flip time to.

Examples

Assign time on flip to the `tStartRefresh` key of `myTimingDict`:

```
win.getTimeOnFlip(myTimingDict, 'tStartRefresh')
```

units

None, ‘height’ (of the window), ‘norm’, ‘deg’, ‘cm’, ‘pix’ Defines the default units of stimuli initialized in the window. I.e. if you change units, already initialized stimuli won’t change their units.

Can be overridden by each stimulus, if units is specified on initialization.

See *Units for the window and stimuli* for explanation of options.

update ()

Deprecated: use `Window.flip()` instead

updateLights (*index=None*)

Explicitly update scene lights if they were modified.

This is required if modifications to objects referenced in *lights* have been changed since assignment. If you removed or added items of *lights* you must refresh all of them.

Parameters index (*int*, *optional*) – Index of light source in *lights* to update. If *None*, all lights will be refreshed.

Examples

Call `updateLights` if you modified lights directly like this:

```
win.lights[1].diffuseColor = [1., 0., 0.]
win.updateLights(1)
```

property `useLights`

Enable scene lighting.

Lights will be enabled if using legacy OpenGL lighting. Stimuli using shaders for lighting should check if `useLights` is `True` since this will have no effect on them, and disable or use a no lighting shader instead. Lights will be transformed to the current view matrix upon setting to `True`.

Lights are transformed by the present `GL_MODELVIEW` matrix. Setting `useLights` will result in their positions being transformed by it. If you want lights to appear at the specified positions in world space, make sure the current matrix defines the view/eye transformation when setting `useLights=True`.

This flag is reset to `False` at the beginning of each frame. Should be `False` if rendering 2D stimuli or else the colors will be incorrect.

property `viewMatrix`

View matrix defined as a 4x4 numpy array.

property `viewPos`

The origin of the window onto which stimulus-objects are drawn.

The value should be given in the units defined for the window. NB: Never change a single component (x or y) of the origin, instead replace the `viewPos`-attribute in one shot, e.g.:

```
win.viewPos = [new_xval, new_yval] # This is the way to do it
win.viewPos[0] = new_xval # DO NOT DO THIS! Errors will result.
```

property `viewport`

Viewport rectangle (x, y, w, h) for the current draw buffer.

Values `x` and `y` define the origin, and `w` and `h` the size of the rectangle in pixels.

This is typically set to cover the whole buffer, however it can be changed for applications like multi-view rendering. Stimuli will draw according to the new shape of the viewport, for instance and stimulus with position (0, 0) will be drawn at the center of the viewport, not the window.

Examples

Constrain drawing to the left and right halves of the screen, where stimuli will be drawn centered on the new rectangle. Note that you need to set both the `viewport` and the `scissor` rectangle:

```
x, y, w, h = win.frameBufferSize # size of the framebuffer
win.viewport = win.scissor = [x, y, w / 2.0, h]
# draw left stimuli ...

win.viewport = win.scissor = [x + (w / 2.0), y, w / 2.0, h]
# draw right stimuli ...

# restore drawing to the whole screen
win.viewport = win.scissor = [x, y, w, h]
```

property `waitBlanking`

After a call to `flip()` should we wait for the blank before the script continues.

property windowedSize

Size of the window to use when not fullscreen (w, h).

9.3.39 Window

A class representing a window for displaying one or more stimuli.

```
class psychopy.visual.Window(size=800, 600, pos=None, color=0, 0, 0, colorSpace='rgb',
    rgb=None, dkl=None, lms=None, fullscr=None, allowGUI=None,
    monitor=None, bitsMode=None, winType=None, units=None,
    gamma=None, blendMode='avg', screen=0, viewScale=None,
    viewPos=None, viewOri=0.0, waitBlanking=True, allowSten-
    cil=False, multiSample=False, numSamples=2, stereo=False,
    name='window1', checkTiming=True, useFBO=False,
    useRetina=True, autoLog=True, gammaErrorPolicy='raise',
    bpc=8, 8, 8, depthBits=8, stencilBits=8, backendConf=None)
```

Used to set up a context in which to draw objects, using either `pyglet`, `pygame`, or `glfw`.

The `pyglet` backend allows multiple windows to be created, allows the user to specify which screen to use (if more than one is available, duh!) and allows movies to be rendered.

The `GLFW` backend is a new addition which provides most of the same features as `pyglet`, but provides greater flexibility for complex display configurations.

`Pygame` may still work for you but it's officially deprecated in this project (we won't be fixing `pygame`-specific bugs).

These attributes can only be set at initialization. See further down for a list of attributes which can be changed after initialization of the `Window`, e.g. `color`, `colorSpace`, `gamma` etc.

Parameters

- **size** (*array-like of int*) – Size of the window in pixels [x, y].
- **pos** (*array-like of int*) – Location of the top-left corner of the window on the screen [x, y].
- **color** (*array-like of float*) – Color of background as [r, g, b] list or single value. Each gun can take values between -1.0 and 1.0.
- **fullscr** (*bool or None*) – Create a window in ‘full-screen’ mode. Better timing can be achieved in full-screen mode.
- **allowGUI** (*bool or None*) – If set to False, window will be drawn with no frame and no buttons to close etc., use *None* for value from preferences.
- **winType** (*str or None*) – Set the window type or back-end to use. If *None* then PsychoPy will revert to user/site preferences.
- **monitor** (*Monitor or None*) – The monitor to be used during the experiment. If *None* a default monitor profile will be used.
- **units** (*str or None*) – Defines the default units of stimuli drawn in the window (can be overridden by each stimulus). Values can be *None*, ‘height’ (of the window), ‘norm’ (normalised), ‘deg’, ‘cm’, ‘pix’. See *Units for the window and stimuli* for explanation of options.
- **screen** (*int*) – Specifies the physical screen that stimuli will appear on (‘`pyglet`’ and ‘`glfw`’ *winType* only). Values can be >0 if more than one screen is present.

- **viewScale** (*array-like of float or None*) – Scaling factors [x, y] to apply custom scaling to the current units of the *Window* instance.
- **viewPos** (*array-like of float or None*) – If not *None*, redefines the origin within the window, in the units of the window. Values outside the borders will be clamped to lie on the border.
- **viewOri** (*float*) – A single value determining the orientation of the view in degrees.
- **waitBlanking** (*bool or None*) – After a call to *flip()* should we wait for the blank before the script continues.
- **bitsMode** (*bool*) – DEPRECATED in 1.80.02. Use *BitsSharp* class from *pycrsld* instead.
- **checkTiming** (*bool*) – Whether to calculate frame duration on initialization. Estimated duration is saved in *monitorFramePeriod*.
- **allowStencil** (*bool*) – When set to *True*, this allows operations that use the OpenGL stencil buffer (notably, allowing the *Aperture* to be used).
- **multiSample** (*bool*) – If *True* and your graphics driver supports multisample buffers, multiple color samples will be taken per-pixel, providing an anti-aliased image through spatial filtering. This setting cannot be changed after opening a window. Only works with ‘pyglet’ and ‘glfw’ *winTypes*, and *useFBO* is *False*.
- **numSamples** (*int*) – A single value specifying the number of samples per pixel if multi-sample is enabled. The higher the number, the better the image quality, but can delay frame flipping. The largest number of samples is determined by *GL_MAX_SAMPLES*, usually 16 or 32 on newer hardware, will crash if number is invalid.
- **stereo** (*bool*) – If *True* and your graphics card supports quad buffers then this will be enabled. You can switch between left and right-eye scenes for drawing operations using *setBuffer()*.
- **useRetina** (*bool*) – In PsychoPy >1.85.3 this should always be *True* as pyglet (or Apple) no longer allows us to create a non-retina display. NB when you use Retina display the initial win size request will be in the larger pixels but subsequent use of *units='pix'* should refer to the tiny Retina pixels. *Window.size* will give the actual size of the screen in Retina pixels.
- **gammaErrorPolicy** (*str*) – If *raise*, an error is raised if the gamma table is unable to be retrieved or set. If *warn*, a warning is raised instead. If *ignore*, neither an error nor a warning are raised.
- **bpc** (*array_like or int*) – Bits per color (BPC) for the back buffer as a tuple to specify bit depths for each color channel separately (red, green, blue), or a single value to set all of them to the same value. Valid values depend on the output color depth of the display (screen) the window is set to use and the system graphics configuration. By default, it is assumed the display has 8-bits per color (8, 8, 8). Behaviour may be undefined for non-fullscreen windows, or if multiple screens are attached with varying color output depths.
- **depthBits** (*int*) – Back buffer depth bits. Default is 8, but can be set higher (eg. 24) if drawing 3D stimuli to minimize artifacts such as ‘Z-fighting’.
- **stencilBits** (*int*) – Back buffer stencil bits. Default is 8.
- **backendConf** (*dict or None*) – Additional options to pass to the backend specified by *winType*. Each backend may provide unique functionality which may not be available across all of them. This allows you to pass special configuration options to a specific backend to configure the feature.

Notes

- Some parameters (e.g. units) can now be given default values in the user/site preferences and these will be used if *None* is given here. If you do specify a value here it will take precedence over preferences.

size

Dimensions of the window's drawing area/buffer in pixels [w, h].

Type array-like (float)

monitorFramePeriod

Refresh rate of the display if `checkTiming=True` on window instantiation.

Type float

__assignFlipTime (*obj, attrib*)

Helper function to assign the time of last flip to the `obj.attrib`

Parameters

- **obj** (*dict or object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of `obj` to assign the flip time to.

__checkMatchingSizes (*requested, actual*)

Checks whether the requested and actual screen sizes differ. If not then a warning is output and the window size is set to actual

__cleanEditables ()

Make sure there are no dead refs in the editables list

__endOfFlip (*clearBuffer*)

Override end of flip with custom color channel masking if required.

__getFrame (*rect=None, buffer='front'*)

Return the current Window as an image.

__getRegionOfFrame (*rect=- 1, 1, 1, - 1, buffer='front', power2=False, squarePower2=False*)

Deprecated function, here for historical reasons. You may now use `:py:attr: ~Window._getFrame()` and specify a `rect` to get a sub-region, just as used here.

`power2` can be useful with older OpenGL versions to avoid interpolation in *PatchStim*. If `power2` or `squarePower2`, it will expand `rect` dimensions up to next power of two. `squarePower2` uses the max dimensions. You need to check what your hardware & OpenGL supports, and call `__getRegionOfFrame()` as appropriate.

__renderFBO ()

Perform a warp operation.

(in this case a copy operation without any warping)

__setCurrent ()

Make this window's OpenGL context current.

If called on a window whose context is current, the function will return immediately. This reduces the number of redundant calls if no context switch is required. If `useFBO=True`, the framebuffer is bound after the context switch.

__setupGL ()

Setup OpenGL state for this window.

`_setupGamma` (*gammaVal*)

A private method to work out how to handle gamma for this Window given that the user might have specified an explicit value, or maybe gave a Monitor.

`_startOfFlip` ()

Custom hardware classes may want to prevent flipping from occurring and can override this method as needed.

Return *True* to indicate hardware flip.

`addEditable` (*editable*)

Adds an editable element to the screen (something to which characters can be sent with meaning from the keyboard).

The current editable object receiving chars is `Window.currentEditable`

Parameters *editable* –

Returns

property `ambientLight`

Ambient light color for the scene [r, g, b, a]. Values range from 0.0 to 1.0. Only applicable if *useLights* is *True*.

Examples

Setting the ambient light color:

```
win.ambientLight = [0.5, 0.5, 0.5]

# don't do this!!!
win.ambientLight[0] = 0.5
win.ambientLight[1] = 0.5
win.ambientLight[2] = 0.5
```

`applyEyeTransform` (*clearDepth=True*)

Apply the current view and projection matrices.

Matrices specified by attributes *viewMatrix* and *projectionMatrix* are applied using ‘immediate mode’ OpenGL functions. Subsequent drawing operations will be affected until *flip()* is called.

All transformations in `GL_PROJECTION` and `GL_MODELVIEW` matrix stacks will be cleared (set to identity) prior to applying.

Parameters `clearDepth` (*bool*) – Clear the depth buffer. This may be required prior to rendering 3D objects.

Examples

Using a custom view and projection matrix:

```
# Must be called every frame since these values are reset after
# `flip()` is called!
win.viewMatrix = viewtools.lookAt( ... )
win.projectionMatrix = viewtools.perspectiveProjectionMatrix( ... )
win.applyEyeTransform()
# draw 3D objects here ...
```

property aspect

Aspect ratio of the current viewport (width / height).

blendMode

Blend mode to use.

callOnFlip (*function, *args, **kwargs*)

Call a function immediately after the next *flip()* command.

The first argument should be the function to call, the following args should be used exactly as you would for your normal call to the function (can use ordered arguments or keyword arguments as normal).

e.g. If you have a function that you would normally call like this:

```
pingMyDevice(portToPing, channel=2, level=0)
```

then you could call *callOnFlip()* to have the function call synchronized with the frame flip like this:

```
win.callOnFlip(pingMyDevice, portToPing, channel=2, level=0)
```

clearBuffer (*color=True, depth=False, stencil=False*)

Clear the present buffer (to which you are currently drawing) without flipping the window.

Useful if you want to generate movie sequences from the back buffer without actually taking the time to flip the window.

Set *color* prior to clearing to set the color to clear the color buffer to. By default, the depth buffer is cleared to a value of 1.0.

Parameters

- **color** (*bool*) – Buffers to clear.
- **depth** (*bool*) – Buffers to clear.
- **stencil** (*bool*) – Buffers to clear.

Examples

Clear the color buffer to a specified color:

```
win.color = (1, 0, 0)
win.clearBuffer(color=True)
```

Clear only the depth buffer, *depthMask* must be *True* or else this will have no effect. Depth mask is usually *True* by default, but may change:

```
win.depthMask = True
win.clearBuffer(color=False, depth=True, stencil=False)
```

close()

Close the window (and reset the Bits++ if necess).

property color

Set the color of the window.

This command sets the color that the blank screen will have on the next clear operation. As a result it effectively takes TWO *flip()* operations to become visible (the first uses the color to create the new screen, the second presents that screen to the viewer). For this reason, if you want to changed background color of the window “on the fly”, it might be a better idea to draw a *Rect* that fills the whole window with the desired *Rect.fillColor* attribute. That’ll show up on first flip.

See other stimuli (e.g. *GratingStim.color*) for more info on the color attribute which essentially works the same on all PsychoPy stimuli.

See *Color spaces* for further information about the ways to specify colors and their various implications.

property colorSpace

The name of the color space currently being used

Value should be: a string or None

For strings and hex values this is not needed. If None the default colorSpace for the stimulus is used (defined during initialisation).

Please note that changing colorSpace does not change stimulus parameters. Thus you usually want to specify colorSpace before setting the color. Example:

```
# A light green text
stim = visual.TextStim(win, 'Color me!',
                      color=(0, 1, 0), colorSpace='rgb')

# An almost-black text
stim.colorSpace = 'rgb255'

# Make it light green again
stim.color = (128, 255, 128)
```

property contentScaleFactor

Scaling factor (*float*) to use when drawing to the backbuffer to convert framebuffer to client coordinates.

See also:

getContentScaleFactor

property convergeOffset

Convergence offset from monitor in centimeters.

This is value corresponds to the offset from screen plane to set the convergence plane (or point for *toe-in* projections). Positive offsets move the plane farther away from the viewer, while negative offsets nearer. This value is used by *setPerspectiveView* and should be set before calling it to take effect.

Notes

- This value is only applicable for *setToeIn* and *setOffAxisView*.

coordToRay (*screenXY*)

Convert a screen coordinate to a direction vector.

Takes a screen/window coordinate and computes a vector which projects a ray from the viewpoint through it (line-of-sight). Any 3D point touching the ray will appear at the screen coordinate.

Uses the current *viewport* and *projectionMatrix* to calculate the vector. The vector is in eye-space, where the origin of the scene is centered at the viewpoint and the forward direction aligned with the -Z axis. A ray of (0, 0, -1) results from a point at the very center of the screen assuming symmetric frustums.

Note that if you are using a flipped/mirrored view, you must invert your supplied screen coordinates (*screenXY*) prior to passing them to this function.

Parameters *screenXY* (*array_like*) – X, Y screen coordinate. Must be in units of the window.

Returns Normalized direction vector [x, y, z].

Return type ndarray

Examples

Getting the direction vector between the mouse cursor and the eye:

```
mx, my = mouse.getPos()
dir = win.coordToRay((mx, my))
```

Set the position of a 3D stimulus object using the mouse, constrained to a plane. The object origin will always be at the screen coordinate of the mouse cursor:

```
# the eye position in the scene is defined by a rigid body pose
win.viewMatrix = camera.getViewMatrix()
win.applyEyeTransform()

# get the mouse location and calculate the intercept
mx, my = mouse.getPos()
ray = win.coordToRay([mx, my])
result = intersectRayPlane( # from mathtools
    orig=camera.pos,
    dir=camera.transformNormal(ray),
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))

# if result is `None`, there is no intercept
if result is not None:
    pos, dist = result
    objModel.thePose.pos = pos
else:
    objModel.thePose.pos = (0, 0, -10) # plane origin
```

If you don't define the position of the viewer with a *RigidBodyPose*, you can obtain the appropriate eye position and rotate the ray by doing the following:

```
pos = numpy.linalg.inv(win.viewMatrix)[:3, 3]
ray = win.coordToRay([mx, my]).dot(win.viewMatrix[:3, :3])
# then ...
result = intersectRayPlane(
    orig=pos,
    dir=ray,
    planeOrig=(0, 0, -10),
    planeNormal=(0, 1, 0))
```

property `cullFace`

True if face culling is enabled.

property `cullFaceMode`

Face culling mode, either *back*, *front* or *both*.

property `currentEditable`

The editable (Text?) object that currently has key focus

property `depthFunc`

Depth test comparison function for rendering.

property `depthMask`

True if depth masking is enabled. Writing to the depth buffer will be disabled.

property depthTest

True if depth testing is enabled.

property draw3d

True if 3D drawing is enabled on this window.

property eyeOffset

Eye offset in centimeters.

This value is used by *setPerspectiveView* to apply a lateral offset to the view, therefore it must be set prior to calling it. Use a positive offset for the right eye, and a negative one for the left. Offsets should be the distance to from the middle of the face to the center of the eye, or half the inter-ocular distance.

property farClip

Distance to the far clipping plane in meters.

flip (*clearBuffer=True*)

Flip the front and back buffers after drawing everything for your frame. (This replaces the *update()* method, better reflecting what is happening underneath).

Parameters **clearBuffer** (*bool, optional*) – Clear the draw buffer after flipping. Default is *True*.

Returns Wall-clock time in seconds the flip completed. Returns *None* if *waitBlanking* is *False*.

Return type *float* or *None*

Notes

- The time returned when *waitBlanking* is *True* corresponds to when the graphics driver releases the draw buffer to accept draw commands again. This time is usually close to the vertical sync signal of the display.

Examples

Results in a clear screen after flipping:

```
win.flip(clearBuffer=True)
```

The screen is not cleared (so represent the previous screen):

```
win.flip(clearBuffer=False)
```

fps ()

Report the frames per second since the last call to this function (or since the window was created if this is first call)

property framebufferSize

Size of the framebuffer in pixels (w, h).

property frontFace

Face winding order to define front, either *ccw* or *cw*.

fullscr

Set whether fullscreen mode is *True* or *False* (not all backends can toggle an open window).

gamma

Set the monitor gamma for linearization.

Warning: Don't use this if using a Bits++ or Bits#, as it overrides monitor settings.

gammaRamp

Sets the hardware CLUT using a specified 3xN array of floats ranging between 0.0 and 1.0.

Array must have a number of rows equal to $2^{\max(\text{bpc})}$.

getActualFrameRate (*nIdentical=10, nMaxFrames=100, nWarmUpFrames=10, threshold=1*)

Measures the actual frames-per-second (FPS) for the screen.

This is done by waiting (for a max of *nMaxFrames*) until *nIdentical* frames in a row have identical frame times (std dev below *threshold* ms).

Parameters

- **nIdentical** (*int, optional*) – The number of consecutive frames that will be evaluated. Higher → greater precision. Lower → faster.
- **nMaxFrames** (*int, optional*) – The maximum number of frames to wait for a matching set of *nIdentical*.
- **nWarmUpFrames** (*int, optional*) – The number of frames to display before starting the test (this is in place to allow the system to settle after opening the *Window* for the first time).
- **threshold** (*int or float, optional*) – The threshold for the std deviation (in ms) before the set are considered a match.

Returns Frame rate (FPS) in seconds. If there is no such sequence of identical frames a warning is logged and *None* will be returned.

Return type *float* or *None*

getContentScaleFactor ()

Get the scaling factor required for scaling correctly on high-DPI displays.

If the returned value is 1.0, no scaling needs to be applied to objects drawn on the backbuffer. A value >1.0 indicates that the backbuffer is larger than the reported client area, requiring points to be scaled to maintain constant size across similarly sized displays. In other words, the scaling required to convert framebuffer to client coordinates.

Returns Scaling factor to be applied along both horizontal and vertical dimensions.

Return type *float*

Examples

Get the size of the client area:

```
clientSize = win.frameBufferSize / win.getContentScaleFactor()
```

Get the framebuffer size from the client size:

```
frameBufferSize = win.clientSize * win.getContentScaleFactor()
```

Convert client (window) to framebuffer pixel coordinates (eg., a mouse coordinate, vertices, etc.):

```
# `mousePosXY` is an array ...
frameBufferXY = mousePosXY * win.getContentScaleFactor()
# you can also use the attribute ...
frameBufferXY = mousePosXY * win.contentScaleFactor
```

Notes

- This value is only valid after the window has been fully realized.

getFutureFlipTime (*targetTime=0, clock=None*)

The expected time of the next screen refresh. This is currently calculated as `win._lastFrameTime + refreshInterval`

Parameters

- **targetTime** (*float*) – The delay *from now* for which you want the flip time. 0 will give the because that the earliest we can achieve. 0.15 will give the schedule flip time that gets as close to 150 ms as possible
- **clock** (*None, 'ptb', 'now' or any Clock object*) – If True then the time returned is compatible with `ptb.GetSecs()`
- **verbose** (*bool*) – Set to True to view the calculations along the way

getMovieFrame (*buffer='front'*)

Capture the current Window as an image.

Saves to stack for `saveMovieFrames()`. As of v1.81.00 this also returns the frame as a PIL image

This can be done at any time (usually after a `flip()` command).

Frames are stored in memory until a `saveMovieFrames()` command is issued. You can issue `getMovieFrame()` as often as you like and then save them all in one go when finished.

The back buffer will return the frame that hasn't yet been 'flipped' to be visible on screen but has the advantage that the mouse and any other overlapping windows won't get in the way.

The default front buffer is to be called immediately after a `flip()` and gives a complete copy of the screen at the window's coordinates.

Parameters **buffer** (*str, optional*) – Buffer to capture.

Returns Buffer pixel contents as a PIL/Pillow image object.

Return type Image

getMsPerFrame (*nFrames=60, showVisual=False, msg="", msDelay=0.0*)

Assesses the monitor refresh rate (average, median, SD) under current conditions, over at least 60 frames.

Records time for each refresh (frame) for n frames (at least 60), while displaying an optional visual. The visual is just eye-candy to show that something is happening when assessing many frames. You can also give it text to display instead of a visual, e.g., `msg='(testing refresh rate...)'`; setting `msg` implies `showVisual == False`.

To simulate refresh rate under cpu load, you can specify a time to wait within the loop prior to doing the `flip()`. If $0 < \text{msDelay} < 100$, wait for that long in ms.

Returns timing stats (in ms) of:

- average time per frame, for all frames
- standard deviation of all frames

- median, as the average of 12 frame times around the median (~monitor refresh rate)

Author

- 2010 written by Jeremy Gray

property `lights`

Scene lights.

This is specified as an array of `~psychopy.visual.LightSource` objects. If a single value is given, it will be converted to a *list* before setting. Set `useLights` to `True` before rendering to enable lighting/shading on subsequent objects. If `lights` is `None` or an empty *list*, no lights will be enabled if `useLights=True`, however, the scene ambient light set with `ambientLight` will still be used.

Examples

Create a directional light source and add it to scene lights:

```
dirLight = gltools.LightSource((0., 1., 0.), lightType='directional')
win.lights = dirLight # `win.lights` will be a list when accessed!
```

Multiple lights can be specified by passing values as a list:

```
myLights = [gltools.LightSource((0., 5., 0.)),
            gltools.LightSource((-2., -2., 0.))]
win.lights = myLights
```

`logOnFlip` (*msg*, *level*, *obj=None*)

Send a log message that should be time-stamped at the next `flip()` command.

Parameters

- **msg** (*str*) – The message to be logged.
- **level** (*int*) – The level of importance for the message.
- **obj** (*object*, *optional*) – The python object that might be associated with this message if desired.

property `mouseVisible`

Sets the visibility of the mouse cursor.

If `Window` was initialized with `allowGUI=False` then the mouse is initially set to invisible, otherwise it will initially be visible.

Usage:

```
win.mouseVisible = False
win.mouseVisible = True
```

property `nearClip`

Distance to the near clipping plane in meters.

property `nextEditable` ()

Moves focus of the cursor to the next editable window

property `projectionMatrix`

Projection matrix defined as a 4x4 numpy array.

recordFrameIntervals

Record time elapsed per frame.

Provides accurate measures of frame intervals to determine whether frames are being dropped. The intervals are the times between calls to `flip()`. Set to `True` only during the time-critical parts of the script. Set this to `False` while the screen is not being updated, i.e., during any slow, non-frame-time-critical sections of your code, including `inter-trial-intervals`, `event.waitkeys()`, `core.wait()`, or `image.setImage()`.

Examples

Enable frame interval recording, successive frame intervals will be stored:

```
win.recordFrameIntervals = True
```

Frame intervals can be saved by calling the `saveFrameIntervals` method:

```
win.saveFrameIntervals()
```

removeEditable (*editable*)

resetEyeTransform (*clearDepth=True*)

Restore the default projection and view settings to PsychoPy defaults. Call this prior to drawing 2D stimuli objects (i.e. `GratingStim`, `ImageStim`, `Rect`, etc.) if any eye transformations were applied for the stimuli to be drawn correctly.

Parameters `clearDepth` (*bool*) – Clear the depth buffer upon reset. This ensures successive draw commands are not affected by previous data written to the depth buffer. Default is `True`.

Notes

- Calling `flip()` automatically resets the view and projection to defaults. So you don't need to call this unless you are mixing 3D and 2D stimuli.

Examples

Going between 3D and 2D stimuli:

```
# 2D stimuli can be drawn before setting a perspective projection
win.setPerspectiveView()
# draw 3D stimuli here ...
win.resetEyeTransform()
# 2D stimuli can be drawn here again ...
win.flip()
```

resetViewport ()

Reset the viewport to cover the whole framebuffer.

Set the viewport to match the dimensions of the back buffer or framebuffer (if `useFBO=True`). The scissor rectangle is also set to match the dimensions of the viewport.

property rgb

saveFrameIntervals (*fileName=None, clear=True*)

Save recorded screen frame intervals to disk, as comma-separated values.

Parameters

- **fileName** (*None* or *str*) – *None* or the filename (including path if necessary) in which to store the data. If *None* then 'lastFrameIntervals.log' will be used.
- **clear** (*bool*) – Clear buffer frames intervals were stored after saving. Default is *True*.

saveMovieFrames (*fileName, codec='libx264', fps=30, clearFrames=True*)

Writes any captured frames to disk.

Will write any format that is understood by PIL (tif, jpg, png, ...)

Parameters

- **filename** (*str*) – Name of file, including path. The extension at the end of the file determines the type of file(s) created. If an image type (e.g. .png) is given, then multiple static frames are created. If it is .gif then an animated GIF image is created (although you will get higher quality GIF by saving PNG files and then combining them in dedicated image manipulation software, such as GIMP). On Windows and Linux .mpeg files can be created if *pymedia* is installed. On macOS .mov files can be created if the *pyobjc-frameworks-UIKit* is installed. Unfortunately the libs used for movie generation can be flaky and poor quality. As for animated GIFs, better results can be achieved by saving as individual .png frames and then combining them into a movie using software like *ffmpeg*.
- **codec** (*str, optional*) – The codec to be used by **moviepy** for mp4/mpg/mov files. If *None* then the default will depend on file extension. Can be one of *libx264*, *mpeg4* for mp4/mov files. Can be *rawvideo*, *png* for avi files (not recommended). Can be *libvorbis* for ogv files. Default is *libx264*.
- **fps** (*int, optional*) – The frame rate to be used throughout the movie. **Only for quicktime (.mov) movies..** Default is *30*.
- **clearFrames** (*bool, optional*) – Set this to *False* if you want the frames to be kept for additional calls to **saveMovieFrames**. Default is *True*.

Examples

Writes a series of static frames as frame001.tif, frame002.tif etc.:

```
myWin.saveMovieFrames('frame.tif')
```

As of PsychoPy 1.84.1 the following are written with **moviepy**:

```
myWin.saveMovieFrames('stimuli.mp4') # codec = 'libx264' or 'mpeg4'
myWin.saveMovieFrames('stimuli.mov')
myWin.saveMovieFrames('stimuli.gif')
```

property scissor

Scissor rectangle (x, y, w, h) for the current draw buffer.

Values *x* and *y* define the origin, and *w* and *h* the size of the rectangle in pixels. The scissor operation is only active if *scissorTest=True*.

Usually, the scissor and viewport are set to the same rectangle to prevent drawing operations from *spilling* into other regions of the screen. For instance, calling *clearBuffer* will only clear within the scissor rectangle.

Setting the scissor rectangle but not the viewport will restrict drawing within the defined region (like a rectangular aperture), not changing the positions of stimuli.

property scissorTest

True if scissor testing is enabled.

property screenshot

setBuffer (*buffer*, *clear=True*)

Choose which buffer to draw to ('left' or 'right').

Requires the Window to be initialised with `stereo=True` and requires a graphics card that supports quad buffering (e.g nVidia Quadro series)

PsychoPy always draws to the back buffers, so 'left' will use `GL_BACK_LEFT`. This then needs to be flipped once both eye's buffers have been rendered.

Parameters

- **buffer** (*str*) – Buffer to draw to. Can either be 'left' or 'right'.
- **clear** (*bool*, *optional*) – Clear the buffer before drawing. Default is `True`.

Examples

Stereoscopic rendering example using quad-buffers:

```
win = visual.Window(..., stereo=True)
while True:
    # clear may not actually be needed
    win.setBuffer('left', clear=True)
    # do drawing for left eye
    win.setBuffer('right', clear=True)
    # do drawing for right eye
    win.flip()
```

setMouseEvent (*name='arrow'*)

Change the appearance of the cursor for this window. Cursor types provide contextual hints about how to interact with on-screen objects.

The graphics used 'standard cursors' provided by the operating system. They may vary in appearance and hot spot location across platforms. The following names are valid on most platforms:

- `arrow`: Default pointer.
- `ibeam`: Indicates text can be edited.
- `crosshair`: Crosshair with hot-spot at center.
- `hand`: A pointing hand.
- `hresize`: Double arrows pointing horizontally.
- `vresize`: Double arrows pointing vertically.

Parameters **name** (*str*) – Type of standard cursor to use (see above). Default is `arrow`.

Notes

- On Windows the `crosshair` option is negated with the background color. It will not be visible when placed over 50% grey fields.

setOffAxisView (*applyTransform=True, clearDepth=True*)

Set an off-axis projection.

Create an off-axis projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. If *eyeOffset* is correct and the view distance and screen size is defined in the monitor configuration, the resulting view will approximate *ortho-stereo* viewing.

The convergence plane can be adjusted by setting *convergeOffset*. By default, the convergence plane is set to the screen plane. Any points on the screen plane will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling *applyEyeTransform()* afterwards.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

setPerspectiveView (*applyTransform=True, clearDepth=True*)

Set the projection and view matrix to render with perspective.

Matrices are computed using values specified in the monitor configuration with the scene origin on the screen plane. Calculations assume units are in meters. If *eyeOffset* $\neq 0$, the view will be transformed laterally, however the frustum shape will remain the same.

Note that the values of *projectionMatrix* and *viewMatrix* will be replaced when calling this function.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling *applyEyeTransform()* afterwards if *False*.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

setToeInView (*applyTransform=True, clearDepth=True*)

Set toe-in projection.

Create a toe-in projection for subsequent rendering calls. Sets the *viewMatrix* and *projectionMatrix* accordingly so the scene origin is on the screen plane. The value of *convergeOffset* will define the convergence point of the view, which is offset perpendicular to the center of the screen plane. Points falling on a vertical line at the convergence point will have zero disparity.

Parameters

- **applyTransform** (*bool*) – Apply transformations after computing them in immediate mode. Same as calling *applyEyeTransform()* afterwards.
- **clearDepth** (*bool, optional*) – Clear the depth buffer.

Notes

- This projection mode is only ‘correct’ if the viewer’s eyes are converged at the convergence point. Due to perspective, this projection introduces vertical disparities which increase in magnitude with eccentricity. Use *setOffAxisView* if you want to display something the viewer can look around the screen comfortably.

property `size`

Size of the drawable area in pixels (w, h).

property `stencilTest`

True if stencil testing is enabled.

`timeOnFlip` (*obj*, *attrib*)

Retrieves the time on the next flip and assigns it to the *attrib* for this *obj*.

Parameters

- **obj** (*dict* or *object*) – A mutable object (usually a dict of class instance).
- **attrib** (*str*) – Key or attribute of *obj* to assign the flip time to.

Examples

Assign time on flip to the `tStartRefresh` key of `myTimingDict`:

```
win.getTimeOnFlip(myTimingDict, 'tStartRefresh')
```

units

None, ‘height’ (of the window), ‘norm’, ‘deg’, ‘cm’, ‘pix’ Defines the default units of stimuli initialized in the window. I.e. if you change units, already initialized stimuli won’t change their units.

Can be overridden by each stimulus, if units is specified on initialization.

See *Units for the window and stimuli* for explanation of options.

`updateLights` (*index=None*)

Explicitly update scene lights if they were modified.

This is required if modifications to objects referenced in *lights* have been changed since assignment. If you removed or added items of *lights* you must refresh all of them.

Parameters **index** (*int*, *optional*) – Index of light source in *lights* to update. If *None*, all lights will be refreshed.

Examples

Call *updateLights* if you modified lights directly like this:

```
win.lights[1].diffuseColor = [1., 0., 0.]
win.updateLights(1)
```

property `useLights`

Enable scene lighting.

Lights will be enabled if using legacy OpenGL lighting. Stimuli using shaders for lighting should check if *useLights* is *True* since this will have no effect on them, and disable or use a no lighting shader instead. Lights will be transformed to the current view matrix upon setting to *True*.

Lights are transformed by the present *GL_MODELVIEW* matrix. Setting *useLights* will result in their positions being transformed by it. If you want lights to appear at the specified positions in world space, make sure the current matrix defines the view/eye transformation when setting *useLights=True*.

This flag is reset to *False* at the beginning of each frame. Should be *False* if rendering 2D stimuli or else the colors will be incorrect.

property viewMatrix

View matrix defined as a 4x4 numpy array.

viewPos

The origin of the window onto which stimulus-objects are drawn.

The value should be given in the units defined for the window. NB: Never change a single component (x or y) of the origin, instead replace the viewPos-attribute in one shot, e.g.:

```
win.viewPos = [new_xval, new_yval] # This is the way to do it
win.viewPos[0] = new_xval # DO NOT DO THIS! Errors will result.
```

property viewport

Viewport rectangle (x, y, w, h) for the current draw buffer.

Values *x* and *y* define the origin, and *w* and *h* the size of the rectangle in pixels.

This is typically set to cover the whole buffer, however it can be changed for applications like multi-view rendering. Stimuli will draw according to the new shape of the viewport, for instance and stimulus with position (0, 0) will be drawn at the center of the viewport, not the window.

Examples

Constrain drawing to the left and right halves of the screen, where stimuli will be drawn centered on the new rectangle. Note that you need to set both the *viewport* and the *scissor* rectangle:

```
x, y, w, h = win.frameBufferSize # size of the framebuffer
win.viewport = win.scissor = [x, y, w / 2.0, h]
# draw left stimuli ...

win.viewport = win.scissor = [x + (w / 2.0), y, w / 2.0, h]
# draw right stimuli ...

# restore drawing to the whole screen
win.viewport = win.scissor = [x, y, w, h]
```

waitBlanking

After a call to *flip()* should we wait for the blank before the script continues.

property windowedSize

Size of the window to use when not fullscreen (w, h).

9.3.40 `psychopy.visual.windowframepack` - Pack multiple monochrome images into RGB frame

Copyright (C) 2014 Allen Institute for Brain Science

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License Version 3 as published by the Free Software Foundation on 29 June 2007. This program is distributed WITHOUT WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR ANY OTHER WARRANTY, EXPRESSED OR IMPLIED. See the GNU General Public License Version 3 for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>

ProjectorFramePacker

class `psychopy.visual.windowframepack.ProjectorFramePacker` (*win*)

Class which packs 3 monochrome images per RGB frame.

Allowing 180Hz stimuli with DLP projectors such as TI LightCrafter 4500.

The class overrides methods of the `visual.Window` class to pack a monochrome image into each RGB channel. PsychoPy is running at 180Hz. The display device is running at 60Hz. The output projector is producing images at 180Hz.

Frame packing can work with any projector which can operate in 'structured light mode' where each RGB channel is presented sequentially as a monochrome image. Most home and office projectors cannot operate in this mode, but projectors designed for machine vision applications typically will offer this feature.

Example usage to use `ProjectorFramePacker`:

```
from psychopy.visual.windowframepack import ProjectorFramePacker
win = Window(monitor='testMonitor', screen=1,
             fullscr=True, useFBO = True)
framePacker = ProjectorFramePacker (win)
```

Parameters `win` : Handle to the window.

endOffFlip (*clearBuffer*)

Mask RGB cyclically after each flip. We ignore `clearBuffer` and just auto-clear after each hardware flip.

startOffFlip ()

Return True if all channels of the RGB frame have been filled with monochrome images, and the associated window should perform a hardware flip

9.3.41 `psychopy.visual.windowwarp` - warping to spherical, cylindrical, or other projections

Copyright (C) 2014 Allen Institute for Brain Science

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License Version 3 as published by the Free Software Foundation on 29 June 2007. This program is distributed WITHOUT WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR ANY OTHER WARRANTY, EXPRESSED OR IMPLIED. See the GNU General Public License Version 3 for more details. You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>

Warper

class psychopy.visual.windowwarp.**Warper** (*win, warp=None, warpfile=None, warpGridsize=300, eyepoint=0.5, 0.5, flipHorizontal=False, flipVertical=False*)

Class to perform warps.

Supports spherical, cylindrical, warpfile, or None (disabled) warps

Warping is a final operation which can be optionally performed on each frame just before transmission to the display. It is useful for perspective correction when the eye to monitor distance is small (say, under 50 cm), or when projecting to domes or other non-planar surfaces.

These attributes define the projection and can be altered dynamically using the `changeProjection()` method.

Parameters `win` : Handle to the window.

warp ['spherical', 'cylindrical', 'warpfile' or *None*] This table gives the main properties of each projection

Warp	eyepoint modifies warp	verticals parallel	horizontal parallel	perspective correct
spherical	y	n	n	y
cylindrical	y	y	n	n
warpfile	n	?	?	?
None	n	y	y	n

warpfile [*None* or filename containing Blender and Paul Bourke] compatible warp definition. (see <http://paulbourke.net/dome/warpingfisheye/>)

warpGridsize [300] Defines the resolution of the warp in both X and Y when not using a warpfile. Typical values would be 64-300 trading off tolerance for jaggies for speed.

eyepoint [[0.5, 0.5] center of the screen] Position of the eye in X and Y as a fraction of the normalized screen width and height. [0,0] is the bottom left of the screen. [1,1] is the top right of the screen.

flipHorizontal: True or False Flip the entire output horizontally. Useful for back projection scenarios.

flipVertical: True or False Flip the entire output vertically. useful if projector is flipped upside down.

Notes

- 1) **The eye distance from the screen is initialized from the** monitor definition.
- 2) **The eye distance can be altered dynamically by changing** 'warper.dist_cm' and then calling `changeProjection()`.

Example usage to create a spherical projection:

```
from psychopy.visual.windowwarp import Warper
win = Window(monitor='testMonitor', screen=1,
             fullscr=True, useFBO = True)
warper = Warper(win,
                warp='spherical',
                warpfile = "",
```

(continues on next page)

(continued from previous page)

```
warpGridsize = 128,
eyepoint = [0.5, 0.5],
flipHorizontal = False,
flipVertical = False)
```

changeProjection (*warp*, *warpfile=None*, *eyepoint=0.5, 0.5*, *flipHorizontal=False*, *flipVertical=False*)

Allows changing the warp method on the fly. Uses the same parameter definitions as constructor.

Window to display all stimuli below.

Windows and display devices:

- *Window* is the main class to display objects
- *Warper* for non-flat projection screens
- *ProjectorFramePacker* for handling displays with ‘structured light mode’ to achieve high framerates
- *Rift* for Oculus Rift support (Windows 64bit only)
- *VisualSystemHD* for NordicNeuralLab’s VisualSystemHD in-scanner display.

Commonly used:

- *ImageStim* to show images
- *TextStim* to show text
- *TextBox2* rewrite of TextStim (faster, editable with more layout options and formatting)

Shapes (all special classes of ShapeStim):

- *ShapeStim* to draw shapes with arbitrary numbers of vertices
- *Rect* to show rectangles
- *Circle* to show circles
- *Polygon* to show polygons
- *Line* to show a line
- *Pie* to show wedges and semi-circles

Images and patterns:

- *ImageStim* to show images
- *SimpleImageStim* to show images without bells and whistles
- *GratingStim* to show gratings
- *RadialStim* to show annulus, a rotating wedge, a checkerboard etc
- *NoiseStim* to show filtered noise patterns of various forms
- *EnvelopeGrating* to generate second-order stimuli (gratings that can have a carrier and envelope)

Multiple stimuli:

- *ElementArrayStim* to show many stimuli of the same type
- *DotStim* to show and control movement of dots

3D shapes, materials, and lighting:

- *LightSource* to define a light source in a scene

- *SceneSkybox* to render a background skybox for VR and 3D scenes
- *BlinnPhongMaterial* to specify a material using the Blinn-Phong lighting model
- *RigidBodyPose* to define poses of objects in 3D space
- *BoundingBox* to define bounding boxes around 3D objects
- *SphereStim* to show a 3D sphere
- *BoxStim* to show 3D boxes and cubes
- *PlaneStim* to show 3D plane
- *ObjMeshStim* to show Wavefront OBJ meshes loaded from files

Other stimuli:

- *MovieStim* to show movies
- *VlcMovieStim* to show movies using VLC
- *Slider* a new improved class to collect ratings
- *RatingScale* to collect ratings
- *CustomMouse* to change the cursor in windows with GUI. OBS: will be deprecated soon

Meta stimuli (stimuli that operate on other stimuli):

- *BufferImageStim* to make a faster-to-show “screenshot” of other stimuli
- *Aperture* to restrict visibility area of other stimuli

Helper functions:

- *filters* for creating grating textures and Gaussian masks etc.
- *visualhelperfunctions* for tests about whether one stimulus contains another
- *unittools* to convert deg<->radians
- *monitorunittools* to convert cm<->pix<->deg etc.
- *colorspacetools* to convert between supported color spaces
- *viewtools* to work with view projections
- *mathtools* to work with vectors, quaternions, and matrices
- *gltools* to work with OpenGL directly (under development)

9.4 psychopy . sound - for playback and recording of sound

The *psychopy.sound* module provides an interface for audio playback and recording devices. It also provides tools for working with audio samples and performing speech-to-text transcription.

9.4.1 Sound - for audio playback

Audio playback is handled by the `Sound` class. currently supports a choice of sound engines: *PTB*, *pyo*, *sounddevice* or *pygame*. You can select which will be used via the *audioLib* preference. `sound.Sound()` will then refer to one of the following backends:

- *SoundPTB*
- `SoundDevice`
- *SoundPyo*
- *SoundPygame*

This preference can be set on a per-experiment basis by importing preferences, and *setting the audioLib option* to use. Audio playback backends vary in performance due to all sorts of factors. Based on testing done by the team and reports from users, their performance can be summarized as follows:

- The *PTB* library has by far the lowest latencies and is strongly recommended (requires 64 bit Python 3.6+)
- The *pyo* library is, in theory, the highest performer, but in practice it has often had issues (at least on MacOS) with crashes and freezing of experiments, or causing them not to finish properly. If those issues aren't affecting your studies then this could be the one for you.
- The *sounddevice* library has performance that appears to be good (although this might be less so in cases where you have complex rendering being done as well because it operates from the same computer core as the main experiment code). It's newer than *pyo* and so more prone to bugs and we haven't yet added microphone support to record your participants.
- The *pygame* backend is the oldest and should work without errors, but has the least good performance. Use it if latencies for your audio don't matter.

Sounds are actually generated by a variety of classes, depending on which “backend” you use (like *pyo* or *sounddevice*) and these different backends can have slightly different attributes, as below. The user should typically do:

```
from psychopy.sound import Sound
```

The class that gets imported will then be an alias of one of the *Sound Classes* described below.

PTB audio latency

PTB brings a number of advantages in terms of latency.

The first is that it has been designed specifically with low-latency playback in mind (rather than, say, on-the-fly mixing and filtering capabilities). Mario Kleiner has worked very hard to get the best out of the drivers available on each operating system and, as a result, with the most aggressive low-latency settings you can get a sound to play in “immediate” mode with typically in the region of 5ms lag and maybe 1ms precision. That's pretty good compared to the other options that have a lag of 20ms upwards and several ms variability.

BUT, on top of that, PTB allows you to *preschedule* your sound to occur at a particular point in time (e.g. when the trigger is due to be sent or when the screen is due to flip) and the PTB engine will then prepare all the buffers ready to go and will also account for the known latencies in the card. With this method the PTB engine is capable of sub-ms *precision* and even sub-ms *lag*!

Of course, *capable* doesn't mean it's happening in your case. It can depend on many things about the local operating system and hardware. You should test it yourself for your kit, but here is an example of a standard Win10 box using built-in audio (not a fancy audio card):

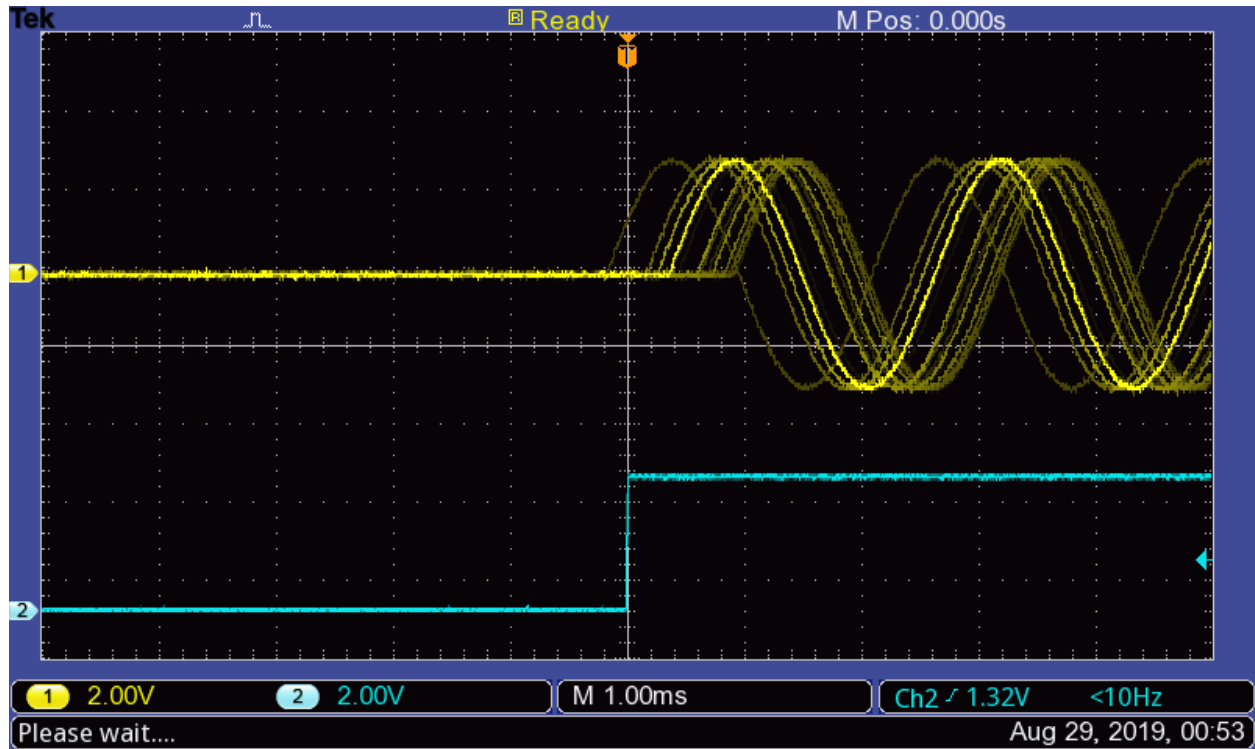


Fig. 9.1: Sub-ms audio timing with standard audio on Win10. Yellow trace is a 440 Hz tone played at 48 kHz with PTB engine. Cyan trace is the trigger (from a Labjack output). Gridlines are set to 1 ms.

Preschedule your sound

The most precise way to use the PTB audio backend is to preschedule the playing of a sound. By doing this PTB can actually take into account both the time taken to load the sound (it will preload ready) and also the time taken by the hardware to start playing it.

To do this you can call `play()` with an argument called `when`. The `when` argument needs to be in the PsychToolBox clock timebase which can be accessed by using `psychtoolbox.GetSecs()` if you want to play sound at an arbitrary time (not in sync with a window flip)

For instance:

```
import psychtoolbox as ptb
from psychopy import sound

mySound = sound.Sound('A')
now = ptb.GetSecs()
mySound.play(when=now+0.5) # play in EXACTLY 0.5s
```

or using `Window.getFutureFlipTime(clock='ptb')` if you want a synchronized time:

```
import psychtoolbox as ptb
from psychopy import sound, visual

mySound = sound.Sound('A')

win = visual.Window()
```

(continues on next page)

(continued from previous page)

```
win.flip()
nextFlip = win.getFutureFlipTime(clock='ptb')

mySound.play(when=nextFlip) # sync with screen refresh
```

The precision of that timing is still dependent on the *PTB Audio Latency Modes* and can obviously not work if the delay before the requested time is not long enough for the requested mode (e.g. if you request that the sound starts on the next refresh but set the latency mode to be 0 (which has a lag of around 300 ms) then the timing will be very poor.

PTB Audio Latency Modes

When using the PTB backend you get the option to choose the Latency Mode, referred to in PsychToolBox as the *reqlatencyclass*.

uses Mode 3 in as a default, assuming that you want low latency and you don't care if other applications can't play sound at the same time (don't listen to iTunes while running your study!)

The modes are as follows:

- 0 : Latency not important** For when it really doesn't matter. Latency can easily be in the region of 300ms! The advantage of this mode is that it will always work and always play a sound, whatever the format of the existing sounds that have been played (with 2, 3, 4 you can obtain low latency but the sampling rate must be the same throughout the experiment).
- 1 : Share low-latency access** Tries to use a low-latency setup in combination with other applications. Latency usually isn't very good and in MS Windows the sound you play must be the same sample rate as any other application that is using the sound system (which means you usually get restricted to exactly 48000 instead of 44100).
- 2 : Exclusive mode low-latency** Takes control of the audio device you're using and dominates it. That can cause some problems for other apps if they're trying to play sounds at the same time.
- 3 : Aggressive exclusive mode** As Mode 2 but with more aggressive settings to prioritise our use of the card over all others. **This is the recommended mode for most studies**
- 4 : Critical mode** As Mode 3 except that, if we fail to be totally dominant, then raise an error rather than just accepting our slightly less dominant status.

Sound Classes

PTB Sound

```
class psychopy.sound.backend_ptb.SoundPTB (value='C', secs=0.5, octave=4, stereo=- 1, volume=1.0, loops=0, sampleRate=None, blockSize=128, preBuffer=- 1, hamming=True, start-Time=0, stopTime=- 1, name="", autoLog=True, syncToWin=None)
```

Play a variety of sounds using the new PsychPortAudio library

Parameters

- **value** – note name (“C”, “Bf”), filename or frequency (Hz)
- **secs** – duration (for synthesised tones)
- **octave** – which octave to use for note names (4 is middle)
- **stereo** – -1 (auto), True or False to force sounds to stereo or mono

- **volume** – float 0-1
- **loops** – number of loops to play (-1=forever, 0=single repeat)
- **sampleRate** – sample rate for synthesized tones
- **blockSize** – the size of the buffer on the sound card (small for low latency, large for stability)
- **preBuffer** – integer to control streaming/buffering - -1 means store all - 0 (no buffer) means stream from disk - potentially we could buffer a few secs(!?)
- **hamming** – boolean (default True) to indicate if the sound should be apodized (i.e., the onset and offset smoothly ramped up from down to zero). The function apodize uses a Hanning window, but arguments named ‘hamming’ are preserved so that existing code is not broken by the change from Hamming to Hanning internally. Not applied to sounds from files.
- **startTime** – for sound files this controls the start of snippet
- **stopTime** – for sound files this controls the end of snippet
- **name** – string for logging purposes
- **autoLog** – whether to automatically log every change
- **syncToWin** – if you want start/stop to sync with win flips add this

_EOS (*reset=True, log=True*)

Function called on End Of Stream

_channelCheck (*array*)

Checks whether stream has fewer channels than data. If True, ValueError

_getDefaultSampleRate ()

Check what streams are open and use one of these

pause ()

Stop the sound but play will continue from here if needed

play (*loops=None, when=None, log=True*)

Start the sound playing

setSound (*value, secs=0.5, octave=4, hamming=None, log=True*)

Set the sound to be played.

Often this is not needed by the user - it is called implicitly during initialisation.

Parameters

value: can be a number, string or an array:

- If it’s a number between 37 and 32767 then a tone will be generated at that frequency in Hz.
- It could be a string for a note (‘A’, ‘Bfl’, ‘B’, ‘C’, ‘Csh’. ...). Then you may want to specify which octave.
- Or a string could represent a filename in the current location, or mediaLocation, or a full path combo
- Or by giving an Nx2 numpy array of floats (-1:1) you can specify the sound yourself as a waveform

secs: duration (only relevant if the value is a note name or a frequency value)

octave: is only relevant if the value is a note name. Middle octave of a piano is 4. Most computers won't output sounds in the bottom octave (1) and the top octave (8) is generally painful

property status

status gives a simple value from psychopy.constants to indicate NOT_STARTED, STARTED, FINISHED, PAUSED

Psychtoolbox sounds also have a statusDetailed property with further info

stop (*reset=True, log=True*)

Stop the sound and return to beginning

property stream

Read-only property returns the the stream on which the sound will be played

property track

The track on the master stream to which we belong

SoundDevice Sound

```
class psychopy.sound.backend_sounddevice.SoundDeviceSound (value='C', secs=0.5,
                                                             octave=4, stereo=- 1,
                                                             volume=1.0, loops=0,
                                                             sampleRate=None,
                                                             blockSize=128,
                                                             preBuffer=- 1, ham-
                                                             ming=True, start-
                                                             Time=0, stopTime=-
                                                             1, name="", au-
                                                             toLog=True)
```

Play a variety of sounds using the new SoundDevice library

Parameters

- **value** – note name (“C”, “Bff”), filename or frequency (Hz)
- **secs** – duration (for synthesised tones)
- **octave** – which octave to use for note names (4 is middle)
- **stereo** – -1 (auto), True or False to force sounds to stereo or mono
- **volume** – float 0-1
- **loops** – number of loops to play (-1=forever, 0=single repeat)
- **sampleRate** – sample rate (for synthesized tones)
- **blockSize** – the size of the buffer on the sound card (small for low latency, large for stability)
- **preBuffer** – integer to control streaming/buffering - -1 means store all - 0 (no buffer) means stream from disk - potentially we could buffer a few secs(!?)
- **hamming** – boolean (default True) to indicate if the sound should be apodized (i.e., the onset and offset smoothly ramped up from down to zero). The function apodize uses a Hanning window, but arguments named ‘hamming’ are preserved so that existing code is not broken by the change from Hamming to Hanning internally. Not applied to sounds from files.
- **startTime** – for sound files this controls the start of snippet

- **stopTime** – for sound files this controls the end of snippet
- **name** – string for logging purposes
- **autoLog** – whether to automatically log every change

_EOS (*reset=True*)

Function called on End Of Stream

_channelCheck (*array*)

Checks whether stream has fewer channels than data. If True, ValueError

pause ()

Stop the sound but play will continue from here if needed

play (*loops=None, when=None*)

Start the sound playing

Parameters when (*not used*) – Included for compatibility purposes

setSound (*value, secs=0.5, octave=4, hamming=None, log=True*)

Set the sound to be played.

Often this is not needed by the user - it is called implicitly during initialisation.

Parameters

value: can be a number, string or an array:

- If it's a number between 37 and 32767 then a tone will be generated at that frequency in Hz.
- It could be a string for a note ('A', 'Bfl', 'B', 'C', 'Csh'. ...). Then you may want to specify which octave.
- Or a string could represent a filename in the current location, or mediaLocation, or a full path combo
- Or by giving an Nx2 numpy array of floats (-1:1) you can specify the sound yourself as a waveform

secs: duration (only relevant if the value is a note name or a frequency value)

octave: is only relevant if the value is a note name. Middle octave of a piano is 4. Most computers won't output sounds in the bottom octave (1) and the top octave (8) is generally painful

stop (*reset=True*)

Stop the sound and return to beginning

property stream

Read-only property returns the the stream on which the sound will be played

Pyo Sound

```
class psychopy.sound.backend_pyo.SoundPyo (value='C', secs=0.5, octave=4, stereo=True,
                                             volume=1.0, loops=0, sampleRate=44100,
                                             bits=16, hamming=True, start=0, stop=- 1,
                                             name="", autoLog=True)
```

Create a sound object, from one of MANY ways.

value: can be a number, string or an array:

- If it's a number between 37 and 32767 then a tone will be generated at that frequency in Hz.
- It could be a string for a note ('A', 'Bfl', 'B', 'C', 'Csh', ...). Then you may want to specify which octave as well
- Or a string could represent a filename in the current location, or mediaLocation, or a full path combo
- Or by giving an Nx2 numpy array of floats (-1:1) you can specify the sound yourself as a waveform

By default, a Hanning window (5ms duration) will be applied to a generated tone, so that onset and offset are smoother (to avoid clicking). To disable the Hanning window, set *hamming=False*.

secs: Duration of a tone. Not used for sounds from a file.

start [float] Where to start playing a sound file; default = 0s (start of the file).

stop [float] Where to stop playing a sound file; default = end of file.

octave: is only relevant if the value is a note name. Middle octave of a piano is 4. Most computers won't output sounds in the bottom octave (1) and the top octave (8) is generally painful

stereo: True (= default, two channels left and right), False (one channel)

volume: loudness to play the sound, from 0.0 (silent) to 1.0 (max). Adjustments are not possible during playback, only before.

loops [int] How many times to repeat the sound after it plays once. If *loops == -1*, the sound will repeat indefinitely until stopped.

sampleRate (= 44100): if the psychopy.sound.init() function has been called or if another sound has already been created then this argument will be ignored and the previous setting will be used

bits: has no effect for the pyo backend

hamming: boolean (default True) to indicate if the sound should be apodized (i.e., the onset and offset smoothly ramped up from down to zero). The function apodize uses a Hanning window, but arguments named 'hamming' are preserved so that existing code is not broken by the change from Hamming to Hanning internally. Not applied to sounds from files.

play (*loops=None, autoStop=True, log=True, when=None*)

Starts playing the sound on an available channel.

loops [int] How many times to repeat the sound after it plays once. If *loops == -1*, the sound will repeat indefinitely until stopped.

when: not used but included for compatibility purposes

For playing a sound file, you cannot specify the start and stop times when playing the sound, only when creating the sound initially.

Playing a sound runs in a separate thread i.e. your code won't wait for the sound to finish before continuing. To pause while playing, you need to use a *psychopy.core.wait(mySound.getDuration())*. If you call *play()* while something is already playing the sounds will be played over each other.

stop (*log=True*)
Stops the sound immediately

pygame Sound

class psychopy.sound.backend_pygame.**SoundPygame** (*value='C', secs=0.5, octave=4, sampleRate=44100, bits=16, name="", autoLog=True, loops=0, stereo=True, hamming=False*)

Create a sound object, from one of many ways.

Parameters

value: can be a number, string or an array:

- If it's a number between 37 and 32767 then a tone will be generated at that frequency in Hz.
- It could be a string for a note ('A', 'Bfl', 'B', 'C', 'Csh', ...). Then you may want to specify which octave as well
- Or a string could represent a filename in the current location, or mediaLocation, or a full path combo
- Or by giving an Nx2 numpy array of floats (-1:1) you can specify the sound yourself as a waveform

secs: duration (only relevant if the value is a note name or a frequency value)

octave: is only relevant if the value is a note name. Middle octave of a piano is 4. Most computers won't output sounds in the bottom octave (1) and the top octave (8) is generally painful

sampleRate(=44100): If a sound has already been created or if the

bits(=16): Pygame uses the same bit depth for all sounds once initialised

Parameters

- **value** (*int, str or array_like*) –
 - If it's a number between 37 and 32767 then a tone will be generated at that frequency in Hz.
 - It could be a string for a note ('A', 'Bfl', 'B', 'C', 'Csh', ...). Then you may want to specify which octave as well.
 - Or a string could represent a filename in the current location, or mediaLocation, or a full path combo.
 - Or by giving an Nx2 numpy array of floats (-1:1) you can specify the sound yourself as a waveform.
- **secs** (*float*) – Duration of sound in seconds (only relevant if the value is a note name or a frequency value).
- **octave** –

fadeOut (*mSecs*)

fades out the sound (when playing) over mSecs. Don't know why you would do this in psychophysics but it's easy and fun to include as a possibility :)

getDuration ()

Gets the duration of the current sound in secs

getVolume ()

Returns the current volume of the sound (0.0:1.0)

play (*fromStart=True, log=True, loops=None, when=None*)

Starts playing the sound on an available channel.

Parameters

fromStart [bool] Not yet implemented.

log [bool] Whether or not to log the playback event.

loops [int] How many times to repeat the sound after it plays once. If *loops* == -1, the sound will repeat indefinitely until stopped.

when: not used but included for compatibility purposes

Notes If no sound channels are available, it will not play and return None. This runs off a separate thread i.e. your code won't wait for the sound to finish before continuing. You need to use a `psychopy.core.wait()` command if you want things to pause. If you call `play()` while something is already playing the sounds will be played over each other.

setVolume (*newVol, log=True*)

Sets the current volume of the sound (0.0:1.0)

stop (*log=True*)

Stops the sound immediately

9.4.2 Microphone - for recording sound

The `Microphone` class provides an interface to audio recording devices connected to the computer. As of now, Psychtoolbox is required to use this feature and must be installed.

Overview

<code>Microphone</code> ([<i>device, sampleRateHz, channels, ...</i>])	Class for recording audio from a microphone or input stream.
--	--

Details

class `psychopy.sound.Microphone` (*device=None, sampleRateHz=None, channels=None, stream-BufferSecs=2.0, maxRecordingSize=24000, policyWhen-Full='warn', audioLatencyMode=None, audioRunMode=0*)

Class for recording audio from a microphone or input stream.

Creating an instance of this class will open a stream using the specified device. Streams should remain open for the duration of your session. When a stream is opened, a buffer is allocated to store samples coming off it. Samples from the input stream will be written to the buffer once `start()` is called.

Parameters

- **device** (int or `~psychopy.sound.AudioDevice`) – Audio capture device to use. You may specify the device either by index (*int*) or descriptor (*AudioDevice*).

- **sampleRateHz** (*int*) – Sampling rate for audio recording in Hertz (Hz). By default, 48kHz (`sampleRateHz=480000`) is used which is adequate for most consumer grade microphones (headsets and built-in).
- **channels** (*int*) – Number of channels to record samples to *1=Mono* and *2=Stereo*.
- **streamBufferSecs** (*float*) – Stream buffer size to pre-allocate for the specified number of seconds. The default is 2.0 seconds which is usually sufficient.
- **maxRecordingSize** (*int*) – Maximum recording size in kilobytes (Kb). Since audio recordings tend to consume a large amount of system memory, one might want to limit the size of the recording buffer to ensure that the application does not run out of memory. By default, the recording buffer is set to 24000 KB (or 24 MB). At a sample rate of 48kHz, this will result in 62.5 seconds of continuous audio being recorded before the buffer is full.
- **audioLatencyMode** (*int or None*) – Audio latency mode to use, values range between 0-4. If *None*, the setting from preferences will be used. Using 3 (exclusive mode) is adequate for most applications and required if using WASAPI on Windows for other settings (such audio quality) to take effect. Symbolic constants `psychoPy.sound.audiodevice.AUDIO_PTB_LATENCY_CLASS_` can also be used.
- **audioRunMode** (*int*) – Run mode for the recording device. Default is standby-mode (0) which allows the system to put the device to sleep. However when the device is needed, waking the device results in some latency. Using a run mode of 1 will keep the microphone running (or 'hot') with reduces latency when the recording is started. Cannot be set when after initialization at this time.

Examples

Capture 10 seconds of audio from the primary microphone:

```
import psychoPy.core as core
import psychoPy.sound.Microphone as Microphone

mic = Microphone(bufferSecs=10.0) # open the microphone
mic.start() # start recording
core.wait(10.0) # wait 10 seconds
mic.stop() # stop recording

audioClip = mic.getRecording()

print(audioClip.duration) # should be ~10 seconds
audioClip.save('test.wav') # save the recorded audio as a 'wav' file
```

The prescribed method for making long recordings is to poll the stream once per frame (or every n-th frame):

```
mic = Microphone(bufferSecs=2.0)
mic.start() # start recording

# main trial drawing loop
mic.poll()
win.flip() # calling the window flip function

mic.stop() # stop recording
audioClip = mic.getRecording()
```

property `audioLatencyMode`

Audio latency mode in use (*int*). Cannot be set after initialization.

bank (*tag=None, transcribe=False, **kwargs*)

Store current buffer as a clip within the microphone object.

This method is used internally by the Microphone component in Builder, don't use it for other applications. Either *stop()* or *pause()* must be called before calling this method.

Parameters

- **tag** (*str* or *None*) – Label for the clip.
- **transcribe** (*bool* or *str*) – Set to the name of a transcription engine (e.g. “GOOGLE”) to transcribe using that engine, or set as *False* to not transcribe.
- **kwargs** (*dict*) – Additional keyword arguments to pass to *transcribe()*.

clear ()

Wipe all clips. Deletes previously banked audio clips.

close ()

Close the stream.

Should not be called until you are certain you're done with it. Ideally, you should never close and reopen the same stream within a single session.

enforceWASAPI = True

flush ()

Get a copy of all banked clips, then clear the clips from storage.

static getDevices ()

Get a *list* of audio capture device (i.e. microphones) descriptors. On Windows, only WASAPI devices are used.

Returns List of *AudioDevice* descriptors for suitable capture devices. If empty, no capture devices have been found.

Return type *list*

getRecording ()

Get audio data from the last microphone recording.

Call this after *stop* to get the recording as an *AudioClip* object. Raises an error if a recording is in progress.

Returns Recorded data between the last calls to *start* (or *record*) and *stop*.

Return type *AudioClip*

property isRecBufferFull

True if there is an overflow condition with the recording buffer.

If this is *True*, then *poll()* is still collecting stream samples but is no longer writing them to anything, causing stream samples to be lost.

property isStarted

True if stream recording has been started (*bool*).

property latencyBias

Latency bias to add when starting the microphone (*float*).

property maxRecordingSize

Maximum recording size in kilobytes (*int*).

Since audio recordings tend to consume a large amount of system memory, one might want to limit the size of the recording buffer to ensure that the application does not run out. By default, the recording buffer

is set to 64000 KB (or 64 MB). At a sample rate of 48kHz, this will result in about. Using stereo audio (`nChannels == 2`) requires twice the buffer over mono (`nChannels == 2`) for the same length clip.

Setting this value will allocate another recording buffer of appropriate size. Avoid doing this in any time sensitive parts of your application.

pause (*blockUntilStopped=True, stopTime=None*)

Pause a recording (alias of `.stop`).

Call this method to end an audio recording if in progress. This will simply halt recording and not close the stream. Any remaining samples will be polled automatically and added to the recording buffer.

Parameters

- **blockUntilStopped** (*bool*) – Halt script execution until the stream has fully stopped.
- **stopTime** (*float or None*) – Scheduled stop time for the stream in system time. If *None*, the stream will stop as soon as possible.

Returns Tuple containing *startTime*, *endPositionSecs*, *xruns* and *estStopTime*.

Return type `tuple`

poll ()

Poll audio samples.

Calling this method adds audio samples collected from the stream buffer to the recording buffer that have been captured since the last *poll* call. Time between calls of this function should be less than *bufferSecs*. You do not need to call this if you call *stop* before the time specified by *bufferSecs* elapses since the *start* call.

Can only be called between called of *start* (or *record*) and *stop* (or *pause*).

Returns Number of overruns in sampling.

Return type `int`

property recBufferSecs

Capacity of the recording buffer in seconds (*float*).

record (*when=None, waitForStart=0, stopTime=None*)

Start an audio recording (alias of `.start()`).

Calling this method will begin capturing samples from the microphone and writing them to the buffer.

Parameters

- **when** (*float, int or None*) – When to start the stream. If the time specified is a floating point (absolute) system time, the device will attempt to begin recording at that time. If *None* or zero, the system will try to start recording as soon as possible.
- **waitForStart** (*bool*) – Wait for sound onset if *True*.
- **stopTime** (*float, int or None*) – Number of seconds to record. If *None* or *-1*, recording will continue forever until *stop* is called.

Returns Absolute time the stream was started.

Return type `float`

property recording

Reference to the current recording buffer (*RecordingBuffer*).

start (*when=None, waitForStart=0, stopTime=None*)

Start an audio recording.

Calling this method will begin capturing samples from the microphone and writing them to the buffer.

Parameters

- **when** (*float, int or None*) – When to start the stream. If the time specified is a floating point (absolute) system time, the device will attempt to begin recording at that time. If *None* or zero, the system will try to start recording as soon as possible.
- **waitForStart** (*bool*) – Wait for sound onset if *True*.
- **stopTime** (*float, int or None*) – Number of seconds to record. If *None* or *-1*, recording will continue forever until *stop* is called.

Returns Absolute time the stream was started.

Return type `float`

property status

Status flag for the microphone. Value can be one of `psychopy.constants.STARTED` or `psychopy.constants.NOT_STARTED`.

For detailed stream status information, use the `streamStatus` property.

stop (*blockUntilStopped=True, stopTime=None*)

Stop recording audio.

Call this method to end an audio recording if in progress. This will simply halt recording and not close the stream. Any remaining samples will be polled automatically and added to the recording buffer.

Parameters

- **blockUntilStopped** (*bool*) – Halt script execution until the stream has fully stopped.
- **stopTime** (*float or None*) – Scheduled stop time for the stream in system time. If *None*, the stream will stop as soon as possible.

Returns Tuple containing *startTime*, *endPositionSecs*, *xruns* and *estStopTime*.

Return type `tuple`

property streamBufferSecs

Size of the internal audio storage buffer in seconds (*float*).

To ensure all data is captured, there must be less time elapsed between subsequent `getAudioClip` calls than `bufferSecs`.

property streamStatus

Status of the audio stream (*AudioDeviceStatus* or *None*).

See `AudioDeviceStatus` for a complete overview of available status fields. This property has a value of *None* if the stream is presently closed.

Examples

Get the capture start time of the stream:

```
# assumes mic.start() was called
captureStartTime = mic.status.captureStartTime
```

Check if microphone recording is active:

```
isActive = mic.status.active
```

Get the number of seconds recorded up to this point:

```
recordedSecs = mic.status.recordedSecs
```

9.4.3 AudioClip - for working with audio data

Overview

<code>AudioClip(samples[, sampleRateHz, userData])</code>	Class for storing audio clip data.
---	------------------------------------

Details

class `psychopy.sound.AudioClip` (*samples, sampleRateHz=48000, userData=None*)

Class for storing audio clip data.

This class is used to store and handle raw audio data, such as those obtained from microphone recordings or loaded from files. PsychoPy stores audio samples in contiguous arrays of 32-bit floating-point values ranging between -1 and 1.

The *AudioClip* class provides basic audio editing capabilities too. You can use operators on *AudioClip* instances to combine audio clips together. For instance, the + operator will return a new *AudioClip* instance whose samples are the concatenation of the two operands:

```
sndCombined = sndClip1 + sndClip2
```

Note that audio clips must have the same sample rates in order to be joined using the addition operator. For online compatibility, use the *append()* method instead.

There are also numerous static methods available to generate various tones (e.g., sine-, saw-, and square-waves). Audio samples can also be loaded and saved to files in various formats (e.g., WAV, FLAC, OGG, etc.)

You can play *AudioClip* by directly passing instances of this object to the *Sound* class:

```
import psychopy.core as core
import psychopy.sound as sound

myTone = AudioClip.sine(duration=5.0) # generate a tone

mySound = sound.Sound(myTone)
mySound.play()
core.wait(5.0) # wait for sound to finish playing
core.quit()
```

Parameters

- **samples** (*ArrayLike*) – Nx1 or Nx2 array of audio samples for mono and stereo, respectively. Values in the array representing the amplitude of the sound waveform should vary between -1 and 1. If not, they will be clipped.
- **sampleRateHz** (*int*) – Sampling rate used to obtain *samples* in Hertz (Hz). The sample rate or frequency is related to the quality of the audio, where higher sample rates usually result in better sounding audio (albeit a larger memory footprint and file size). The value specified should match the frequency the clip was recorded at. If not, the audio may sound distorted when played back. Usually, a sample rate of 48kHz is acceptable for most applications (DVD audio quality). For convenience, module level constants with form `SAMPLE_RATE_*` are provided to specify many common samples rates.
- **userData** (*dict or None*) – Optional user data to associated with the audio clip.

static `_checkCodecSupported` (*codec, raiseError=False*)

Check if the audio format string corresponds to a supported codec. Used internally to check if the user specified a valid codec identifier.

Parameters

- **codec** (*str*) – Codec identifier (e.g., ‘wav’, ‘mp3’, etc.)
- **raiseError** (*bool*) – Raise an error (‘’) instead of returning a value. Default is *False*.

Returns *True* if the format is supported.

Return type *bool*

append (*clip*)

Append samples from another sound clip to the end of this one.

The *AudioClip* object must have the same sample rate and channels as this object.

Parameters **clip** (*AudioClip*) – Audio clip to append.

Returns This object with samples from *clip* appended.

Return type *AudioClip*

Examples

Join two sound clips together:

```
snd1.append(snd2)
```

asMono (*copy=True*)

Convert the audio clip to mono (single channel audio).

Parameters **copy** (*bool*) – If *True* an *AudioClip* containing a copy of the samples will be returned. If *False*, channels will be mixed inplace resulting a the same object being returned. User data is not copied.

Returns Mono version of this object.

Return type *AudioClip*

property channels

Number of audio channels in the clip (*int*).

If *channels* > 1, the audio clip is in stereo.

convertToWAV ()

Get a copy of stored audio samples in WAV PCM format.

Returns Array with the same shapes as *.samples* but in 16-bit WAV PCM format.

Return type ndarray

copy ()

Create an independent copy of this *AudioClip*.

Returns

Return type *AudioClip*

property duration

The duration of the audio in seconds (*float*).

This value is computed using the specified sampling frequency and number of samples.

gain (factor, channel=None)

Apply gain the audio samples.

This will modify the internal store of samples inplace. Clipping is automatically applied to samples after applying gain.

Parameters

- **factor** (*float* or *int*) – Gain factor to multiply audio samples.
- **channel** (*int* or *None*) – Channel to apply gain to. If *None*, gain will be applied to all channels.

property isMono

True if there is only one channel of audio data.

property isStereo

True if there are two channels of audio samples.

Usually one for each ear. The first channel is usually the left ear, and the second the right.

static load (filename, codec=None)

Load audio samples from a file. Note that this is a static method!

Parameters

- **filename** (*str*) – File name to load.
- **codec** (*str* or *None*) – Codec to use. If *None*, the format will be implied from the file name.

Returns Audio clip containing samples loaded from the file.

Return type *AudioClip*

rms (channel=None)

Compute the root mean square (RMS) of the samples to determine the average signal level.

Parameters **channel** (*int* or *None*) – Channel to compute RMS (zero-indexed). If *None*, the RMS of all channels will be computed.

Returns An array of RMS values for each channel if *channel=None* (even if there is one channel an array is returned). If *channel* was specified, a *float* will be returned indicating the RMS of that single channel.

Return type ndarray or *float*

property sampleRateHz

Sample rate of the audio clip in Hz (*int*). Should be the same value as the rate *samples* was captured at.

property samples

Nx1 or Nx2 array of audio samples (*~numpy.ndarray*).

Values must range from -1 to 1. Values outside that range will be clipped, possibly resulting in distortion.

save (*filename, codec=None*)

Save an audio clip to file.

Parameters

- **filename** (*str*) – File name to write audio clip to.
- **codec** (*str or None*) – Format to save audio clip data as. If *None*, the format will be implied from the extension at the end of *filename*.

static sawtooth (*duration=1.0, freqHz=440, peak=1.0, gain=0.8, sampleRateHz=48000, channels=2*)

Generate audio samples for a tone with a sawtooth waveform.

Parameters

- **duration** (*float or int*) – Length of the sound in seconds.
- **freqHz** (*float or int*) – Frequency of the tone in Hertz (Hz). Note that this differs from the *sampleRateHz*.
- **peak** (*float*) – Location of the peak between 0.0 and 1.0. If the peak is at 0.5, the resulting wave will be triangular. A value of 1.0 will cause the peak to be located at the very end of a cycle.
- **gain** (*float*) – Gain factor ranging between 0.0 and 1.0. Default is 0.8.
- **sampleRateHz** (*int*) – Samples rate of the audio for playback.
- **channels** (*int*) – Number of channels for the output.

Returns

Return type *AudioClip*

static silence (*duration=1.0, sampleRateHz=48000, channels=2*)

Generate audio samples for a silent period.

This is used to create silent periods of a very specific duration between other audio clips.

Parameters

- **duration** (*float or int*) – Length of the sound in seconds.
- **sampleRateHz** (*int*) – Samples rate of the audio for playback.
- **channels** (*int*) – Number of channels for the output.

Returns

Return type *AudioClip*

Examples

Generate 5 seconds of silence to enjoy:

```
import psychopy.sound as sound
silence = sound.AudioClip.silence(10.)
```

Use the silence as a break between two audio clips when concatenating them:

```
fullClip = clip1 + sound.AudioClip.silence(10.) + clip2
```

static sine (*duration=1.0, freqHz=440, gain=0.8, sampleRateHz=48000, channels=2*)

Generate audio samples for a tone with a sine waveform.

Parameters

- **duration** (*float or int*) – Length of the sound in seconds.
- **freqHz** (*float or int*) – Frequency of the tone in Hertz (Hz). Note that this differs from the *sampleRateHz*.
- **gain** (*float*) – Gain factor ranging between 0.0 and 1.0. Default is 0.8.
- **sampleRateHz** (*int*) – Samples rate of the audio for playback.
- **channels** (*int*) – Number of channels for the output.

Returns

Return type *AudioClip*

Examples

Generate an audio clip of a tone 10 seconds long with a frequency of 400Hz:

```
import psychopy.sound as sound
tone400Hz = sound.AudioClip.sine(10., 400.)
```

Create a marker/cue tone and append it to pre-recorded instructions:

```
import psychopy.sound as sound
voiceInstr = sound.AudioClip.load('/path/to/instructions.wav')
markerTone = sound.AudioClip.sine(
    1.0, 440., # duration and freq
    sampleRateHz=voiceInstr.sampleRateHz) # must be the same!

fullInstr = voiceInstr + markerTone # create instructions with cue
fullInstr.save('/path/to/instructions_with_tone.wav') # save it
```

static square (*duration=1.0, freqHz=440, dutyCycle=0.5, gain=0.8, sampleRateHz=48000, channels=2*)

Generate audio samples for a tone with a square waveform.

Parameters

- **duration** (*float or int*) – Length of the sound in seconds.
- **freqHz** (*float or int*) – Frequency of the tone in Hertz (Hz). Note that this differs from the *sampleRateHz*.
- **dutyCycle** (*float*) – Duty cycle between 0.0 and 1.0.

- **gain** (*float*) – Gain factor ranging between 0.0 and 1.0. Default is 0.8.
- **sampleRateHz** (*int*) – Samples rate of the audio for playback.
- **channels** (*int*) – Number of channels for the output.

Returns

Return type *AudioClip*

transcribe (*engine='sphinx', language='en-US', expectedWords=None, config=None*)

Convert speech in audio to text.

This feature passes the audio clip samples to a specified text-to-speech engine which will attempt to transcribe any speech within. The efficacy of the transcription depends on the engine selected, audio quality, and language support. By default, Pocket Sphinx is used which provides decent transcription capabilities offline for English and a few other languages. For more robust transcription capabilities with a greater range of language support, online providers such as Google may be used.

Speech-to-text conversion blocks the main application thread when used on Python. Don't transcribe audio during time-sensitive parts of your experiment! This issue is known to the developers and will be fixed in a later release.

Parameters

- **engine** (*str*) – Speech-to-text engine to use. Can be one of 'sphinx' for CMU Pocket Sphinx or 'google' for Google Cloud.
- **language** (*str*) – BCP-47 language code (eg., 'en-US'). Note that supported languages vary between transcription engines.
- **expectedWords** (*list or tuple*) – List of strings representing expected words or phrases. This will constrain the possible output words to the ones specified. Note not all engines support this feature (only Sphinx and Google Cloud do at this time). A warning will be logged if the engine selected does not support this feature. CMU PocketSphinx has an additional feature where the sensitivity can be specified for each expected word. You can indicate the sensitivity level to use by putting a : (colon) after each word in the list (see the Example below). Sensitivity levels range between 0 and 100. A higher number results in the engine being more conservative, resulting in a higher likelihood of false rejections. The default sensitivity is 80% for words/phrases without one specified.
- **config** (*dict or None*) – Additional configuration options for the specified engine. These are specified using a dictionary (ex. *config={'pfilter': 1}* will enable the profanity filter when using the 'google' engine).

Returns Transcription result.

Return type *TranscriptionResult*

Notes

- Online transcription services (eg., Google) provide robust and accurate speech recognition capabilities with broader language support than offline solutions. However, these services may require a paid subscription to use, reliable broadband internet connections, and may not respect the privacy of your participants as their responses are being sent to a third-party. Also consider that a track of audio data being sent over the network can be large, users on metered connections may incur additional costs to run your experiment.
- If the audio clip has multiple channels, they will be combined prior to being passed to the transcription service if needed.

property `userData`

User data associated with this clip (*dict*). Can be used for storing additional data related to the clip. Note that *userData* is not saved with audio files!

Example

Adding fields to *userData*. For instance, we want to associated the start time the clip was recorded at with it:

```
myClip.userData['date_recorded'] = t_start
```

We can access that field later by:

```
thisRecordingStartTime = myClip.userData['date_recorded']
```

static `whiteNoise` (*duration=1.0, sampleRateHz=48000, channels=2*)

Generate gaussian white noise.

New feature, use with caution.

Parameters

- **duration** (*float or int*) – Length of the sound in seconds.
- **sampleRateHz** (*int*) – Samples rate of the audio for playback.
- **channels** (*int*) – Number of channels for the output.

Returns

Return type *AudioClip*

9.4.4 `AudioDeviceInfo` and `AudioDeviceStatus` - descriptors for audio devices

These classes are used to store information about audio devices and their status. Only a subset of PsychoPy’s sound API currently use these classes, such as the *psychopy.sound.Microphone* class.

Overview

<code>AudioDeviceInfo</code> ([<i>deviceIndex</i> , <i>deviceName</i> , ...])	Descriptor for an audio device (playback or recording) on this system.
<code>AudioDeviceStatus</code> ([<i>active</i> , <i>state</i> , ...])	Descriptor for audio device status information.

Details

class `psychopy.sound.AudioDeviceInfo` (*deviceIndex=- 1, deviceName='Null Device', hostAP- IName='Null Audio Driver', outputChannels=0, output- Latency=0.0, 0.0, inputChannels=0, inputLatency=0.0, 0.0, defaultSampleRate=48000, audioLib=""*)

Descriptor for an audio device (playback or recording) on this system.

Properties associated with this class provide information about a specific audio playback or recording device. An object can be then passed to `Microphone` to open a stream using the device described by the object.

This class is usually instanced only by calling `getDevices()`. Users should avoid creating instances of this

class themselves unless they have good reason to.

Parameters

- **deviceIndex** (*int*) – Enumerated index of the audio device. This number is specific to the engine used for audio.
- **deviceName** (*str*) – Human-readable name of the device.
- **hostAPIName** (*str*) – Human-readable name of the host API used for audio.
- **outputChannels** (*int*) – Number of output channels.
- **outputLatency** (*tuple*) – Low (*float*) and high (*float*) output latency in milliseconds.
- **inputChannels** (*int*) – Number of input channels.
- **inputLatency** (*tuple*) – Low (*float*) and high (*float*) input latency in milliseconds.
- **defaultSampleRate** (*int*) – Default sample rate for the device in Hertz (Hz).
- **audioLib** (*str*) – Audio library that queried device information used to populate the properties of this descriptor (e.g., 'ptb' for Psychtoolbox).

Examples

Get a list of available devices:

```
import psychopy.sound as sound
recordingDevicesList = sound.Microphone.getDevices()
```

Get the low and high input latency of the first recording device:

```
recordingDevice = recordingDevicesList[0] # assume not empty
inputLatencyLow, inputLatencyHigh = recordingDevice.inputLatency
```

Get the device name as it may appear in the system control panel or sound settings:

```
deviceName = recordingDevice.deviceName
```

Specifying the device to use for capturing audio from a microphone:

```
# get the first suitable capture device found by the sound engine
recordingDevicesList = sound.Microphone.getDevices()
recordingDevice = recordingDevicesList[0]

# pass the descriptor to microphone to configure it
mic = sound.Microphone(device=recordingDevice)
mic.start() # start recording sound
```

property **audioLib**

Audio library used to query device information (*str*).

static **createFromPTBDesc** (*desc*)

Create an *AudioDevice* instance with values populated using a descriptor (*dict*) returned from the PTB *audio.get_devices* API call.

Parameters *desc* (*dict*) – Audio device descriptor returned from Psychtoolbox's *get_devices* function.

Returns Audio device descriptor with properties set using *desc*.

Return type *AudioDeviceInfo*

property defaultSampleRate

Default sample rate in Hertz (Hz) for this device (*int*).

property deviceIndex

Enumerated index (*int*) of the audio device.

property deviceName

Human-readable name (*str*) for the audio device reported by the driver.

property hostAPIName

Human-readable name (*str*) for the host API.

property inputChannels

Number of input channels (*int*). If >0, this is likely a audio capture device.

property inputLatency

Low and high input latency in milliseconds (*low, high*).

property isCapture

True if this device is suitable for capture (*bool*).

property isDuplex

True if this device is suitable for capture and playback (*bool*).

property isPlayback

True if this device is suitable for playback (*bool*).

property outputChannels

Number of output channels (*int*). If >0, this is likely a audio playback device.

property outputLatency

Low and high output latency in milliseconds (*low, high*).

```
class psychopy.sound.AudioDeviceStatus (active=0, state=0, requestedStartTime=0.0, start-
Time=0.0, captureStartTime=0.0, requestedStop-
Time=0.0, estimatedStopTime=0.0, currentStream-
Time=0.0, elapsedOutSamples=0, positionSecs=0.0,
recordedSecs=0.0, readSecs=0.0, schedulePosi-
tion=0.0, xRuns=0, totalCalls=0, timeFailed=0,
bufferSize=0, cpuLoad=0.0, predictedLatency=0.0,
latencyBias=0.0, sampleRate=48000, outDeviceIn-
dex=0, inDeviceIndex=0, audioLib='Null Audio
Library')
```

Descriptor for audio device status information.

Properties of this class are standardized on the status information returned by Psychtoolbox. Other audio back-ends should try to populate these fields as best they can with their equivalent status values.

Users should never instance this class themselves unless they have good reason to.

Parameters

- **active** (*bool*) – *True* if playback or recording has started, else *False*.
- **state** (*int*) – State of the device, either 1 for playback, 2 for recording or 3 for duplex (recording and playback).
- **requestedStartTime** (*float*) – Requested start time of the audio stream after the start of playback or recording.
- **startTime** (*float*) – The actual (real) start time of audio playback or recording.

- **captureStartTime** (*float*) – Estimate of the start time of audio capture. Only valid if audio capture is active. Usually, this time corresponds to the time when the first sound was captured.
- **requestedStopTime** (*float*) – Stop time requested when starting the stream.
- **estimatedStopTime** (*float*) – Estimated stop time given *requestedStopTime*.
- **currentStreamTime** (*float*) – Estimate of the time it will take for the most recently submitted sample to reach the speaker. Value is in absolute system time and reported for playback only.
- **elapsedOutSamples** (*int*) – Total number of samples submitted since the start of playback.
- **positionSecs** (*float*) – Current stream playback position in seconds this loop. Does not account for hardware or driver latency.
- **recordedSecs** (*float*) – Total amount of recorded sound data (in seconds) since start of capture.
- **readSecs** (*float*) – Total amount of sound data in seconds that has been fetched from the internal buffer.
- **schedulePosition** (*float*) – Current position in a running schedule in seconds.
- **xRuns** (*int*) – Number of dropouts due to buffer over- and under-runs. Such conditions can result in glitches during playback/recording. Even if the number remains zero, that does not mean that glitches did not occur.
- **totalCalls** (*int*) – **Debug** - Used for debugging the audio engine.
- **timeFailed** (*float*) – **Debug** - Used for debugging the audio engine.
- **bufferSize** (*int*) – **Debug** - Size of the buffer allocated to contain stream samples. Used for debugging the audio engine.
- **cpuLoad** (*float*) – Amount of load on the CPU imparted by the sound engine. Ranges between 0.0 and 1.0 where 1.0 indicates maximum load on the core running the sound engine process.
- **predictedLatency** (*float*) – Latency for the given hardware and driver. This indicates how far ahead you need to start the device to ensure it starts at a scheduled time.
- **latencyBias** (*float*) – Additional latency bias added by the user.
- **sampleRate** (*int*) – Sample rate in Hertz (Hz) the playback/recording is using.
- **outDeviceIndex** (*int*) – Enumerated index of the output device.
- **inDeviceIndex** (*int*) – Enumerated index of the input device.
- **audioLib** (*str*) – Identifier for the audio library which created this status.

property active

True if playback or recording has started (*bool*).

property audioLib

Identifier for the audio library which created this status (*str*).

property bufferSize

Debug - Size of the buffer allocated to contain stream samples. Used for debugging the audio engine.

property captureStartTime

Estimate of the start time of audio capture (*float*). Only valid if audio capture is active. Usually, this time corresponds to the time when the first sound was captured.

property cpuLoad

Amount of load on the CPU imparted by the sound engine (*float*). Ranges between 0.0 and 1.0 where 1.0 indicates maximum load on the core running the sound engine process.

static createFromPTBDesc (*desc*)

Create an *AudioDeviceStatus* instance using a status descriptor returned by Psychtoolbox.

Parameters *desc* (*dict*) – Audio device status descriptor.

Returns Audio device descriptor with properties set using *desc*.

Return type *AudioDeviceStatus*

property currentStreamTime

Estimate of the time it will take for the most recently submitted sample to reach the speaker (*float*). Value is in absolute system time and reported for playback mode only.

property elapsedOutSamples

Total number of samples submitted since the start of playback (*int*).

property estimatedStopTime

Estimated stop time given *requestedStopTime* (*float*).

property inDeviceIndex

Enumerated index of the input device (*int*).

property isCapture

True if this device is operating in capture mode (*bool*).

property isDuplex

True if this device is operating capture and recording mode (*bool*).

property isPlayback

True if this device is operating in playback mode (*bool*).

property latencyBias

Additional latency bias added by the user (*float*).

property outDeviceIndex

Enumerated index of the output device (*int*).

property positionSecs

Current stream playback position in seconds this loop (*float*). Does not account for hardware or driver latency.

property predictedLatency

Latency for the given hardware and driver (*float*). This indicates how far ahead you need to start the device to ensure it starts at a scheduled time.

property readSecs

Total amount of sound data in seconds that has been fetched from the internal buffer (*float*).

property recordedSecs

Total amount of recorded sound data (in seconds) since start of capture (*float*).

property requestedStartTime

Requested start time of the audio stream after the start of playback or recording (*float*).

property requestedStopTime

Stop time requested when starting the stream (*float*).

property sampleRate

Sample rate in Hertz (Hz) the playback recording is using (*int*).

property schedulePosition

Current position in a running schedule in seconds (*float*).

property startTime

The actual (real) start time of audio playback or recording (*float*).

property state

State of the device (*int*). Either 1 for playback, 2 for recording or 3 for duplex (recording and playback).

property timeFailed

Debug - Used for debugging the audio engine (*float*).

property totalCalls

Debug - Used for debugging the audio engine (*int*).

property xRuns

Number of dropouts due to buffer over- and under-runs (*int*). Such conditions can result in glitches during playback/recording. Even if the number remains zero, that does not mean that glitches did not occur.

9.5 psychopy.hardware - hardware interfaces

can access a wide range of external hardware. For some devices the interface has already been created in the following sub-packages of . For others you may need to write the code to access the serial port etc. manually.

Contents:

9.5.1 Keyboard

To handle input from keyboard (supersedes `event.getKeys`)

The Keyboard class was new in PsychoPy 3.1 and replaces the older `event.getKeys()` calls.

Psychtoolbox versus `event.getKeys`

On 64 bits Python3 installations it provides access to the Psychtoolbox `kbQueue` series of functions using the same compiled C code (available in `python-psychtoolbox` lib).

On 32 bit installations and Python2 it reverts to the older `psychopy.event.getKeys()` calls.

The new calls have several advantages:

- the polling is performed and timestamped asynchronously with the main thread so that times relate to when the key was pressed, not when the call was made
- the polling is direct to the USB HID library in C, which is faster than waiting for the operating system to poll and interpret those same packets
- we also detect the KeyUp events and therefore provide the option of returning keypress duration
- on Linux and Mac you can also distinguish between different keyboard devices (see `getKeyboards()`)

This library makes use, where possible of the same low-level asynchronous hardware polling as in Psychtoolbox

Example usage

```

from psychopy.hardware import keyboard
from psychopy import core

kb = keyboard.Keyboard()

# during your trial
kb.clock.reset() # when you want to start the timer from
keys = kb.getKeys(['right', 'left', 'quit'], waitRelease=True)
if 'quit' in keys:
    core.quit()
for key in keys:
    print(key.name, key.rt, key.duration)

```

Classes and functions

class psychopy.hardware.keyboard.**Keyboard** (*device=-1, bufferSize=10000, waitForStart=False, clock=None, backend=None*)

The Keyboard class provides access to the Psychtoolbox KbQueue-based calls on **Python3 64-bit** with fall-back to *event.getKeys* on legacy systems.

Create the device (default keyboard or select one)

Parameters

- **device** (*int or dict*) – On Linux/Mac this can be a device index or a dict containing the device info (as from *getKeyboards()*) or -1 for all devices acting as a unified Keyboard
- **bufferSize** (*int*) – How many keys to store in the buffer (before dropping older ones)
- **waitForStart** (*bool (default False)*) – Normally we’ll start polling the Keyboard at all times but you could choose not to do that and start/stop manually instead by setting this to True

classmethod *getBackend* ()

Return backend being used.

getKeys (*keyList=None, waitRelease=True, clear=True*)

Parameters

- **keyList** (*list (or other iterable)*) – The keys that you want to listen out for. e.g. ['left', 'right', 'q']
- **waitRelease** (*bool (default True)*) – If True then we won’t report any “incomplete” keypress but all presses will then be given a *duration*. If False then all keys will be presses will be returned, but only those with a corresponding release will contain a *duration* value (others will have *duration=None*)
- **clear** (*bool (default True)*) – If False then keep the keypresses for further calls (leave the buffer untouched)

Returns

Return type A list of *Keypress* objects

classmethod *setBackend* (*backend*)

Set backend event handler. Returns currently active handler.

Parameters **backend** – ‘iohub’, ‘ptb’, ‘event’, or ‘

Returns str

start ()

Start recording from this keyboard

stop ()

Start recording from this keyboard

waitKeys (*maxWait=inf, keyList=None, waitRelease=True, clear=True*)

Same as `~psychopy.hardware.keyboard.Keyboard.getKeys`, but halts everything (including drawing) while awaiting keyboard input.

Parameters

maxWait [any numeric value.] Maximum number of seconds period and which keys to wait for. Default is float('inf') which simply waits forever.

keyList [None or []] Allows the user to specify a set of keys to check for. Only keypresses from this set of keys will be removed from the keyboard buffer. If the keyList is *None*, all keys will be checked and the key buffer will be cleared completely. NB, pygame doesn't return timestamps (they are always 0)

waitRelease: True or False If True then we won't report any "incomplete" keypress but all presses will then be given a *duration*. If False then all keys will be presses will be returned, but only those with a corresponding release will contain a *duration* value (others will have *duration=None*)

clear [True or False] Whether to clear the keyboard event buffer (and discard preceding keypresses) before starting to monitor for new keypresses.

Returns None if times out.

class `psychopy.hardware.keyboard.KeyPress` (*code, tDown, name=None*)

Class to store key presses, as returned by `Keyboard.getKeys()`

Unlike keypresses from the old `event.getKeys()` which returned a list of strings (the names of the keys) we now return several attributes for each key:

.name: the name as a string (matching the previous pygame name) .rt: the reaction time (relative to last clock reset) .tDown: the time the key went down in absolute time .duration: the duration of the keypress (or None if not released)

Although the keypresses are a class they will test `==`, `!=` and `in` based on their name. So you can still do:

```

kb = KeyBoard()
# wait for keypresses here
keys = kb.getKeys()
for thisKey in keys:
    if thisKey=='q': # it is equivalent to the string 'q'
        core.quit()
    else:
        print(thisKey.name, thisKey.tDown, thisKey.rt)
    
```

`psychopy.hardware.keyboard.getKeyboards` ()

Get info about the available keyboards.

Only really useful on Mac/Linux because on these the info can be used to select a particular physical device when calling `Keyboard`. On Win this function does return information correctly but the `:class:Keyboard` can't make use of it.

Returns USB Info including with name, manufacturer, id, etc for each device

Return type A list of dicts

9.5.2 BrainProducts

Python support for Brain Products GMBH hardware.

Here we have implemented support for the Remote Control Server application, which allows you to control recordings, send annotations etc. all from Python.

```
class psychopy.hardware.brainproducts.RemoteControlServer (host='127.0.0.1',
                                                            port=6700,      time-
                                                            out=1.0,        test-
                                                            Mode=False)
```

Provides a remote-control interface to BrainProducts Recorder.

Example usage:

```
import time
from psychopy import logging
from psychopy.hardware import brainproducts

logging.console.setLevel(logging.DEBUG)
rcs = brainproducts.RemoteControlServer()
rcs.open('testExp',
        workspace='C:/Vision/Workfiles/Standard Workspace.rwksp',
        participant='S0021')
rcs.openRecorder()
time.sleep(2)
rcs.mode = 'monitor' # or 'impedance', or 'default'
rcs.startRecording()
time.sleep(2)
rcs.sendAnnotation('124', 'STIM')
time.sleep(1)
rcs.pauseRecording()
time.sleep(1)
rcs.resumeRecording()
time.sleep(1)
rcs.stopRecording()
time.sleep(1)
rcs.mode = 'default' # stops monitoring mode
```

To initialize the remote control recorder.

Parameters

- **host** (*string, optional*) – The IP address or hostname of the computer running RCS. Defaults to 127.0.0.1.
- **port** (*int, optional*) – The port on which RCS is listening for a connection on the EEG computer. This should usually not need to be changed. Defaults to 6700.
- **timeout** (*float, optional*) – The timeout (in seconds) to wait for sending/receivign commands
- **testMode** (*bool, optional*) – If True, the network connection to the RCS computer will not actually be initialized. Defaults to False.

property amplifier

Get/set the amplifier to use. Could be one of ” [‘actiCHamp’, ‘BrainAmp Family’,”” ‘LiveAmp’, ‘Quick-Amp USB’, ‘Simulated Amplifier’,”” ‘V-Amp / FirstAmp’]

For Liveamp you should also provide the serial number, comma separated from the amplifier type.

Examples

```
rsc = RemoteControlServer() rsc.amplifier = 'LiveAmp', 'LA-05490-0200' # OR rsc.amplifier = 'ac-  
tiCHamp'
```

close()

Closes the recording and deletes all associated workspace variables (e.g. when a participant has been completed)

dcReset()

Use this to reset any DC offset that might have accumulated if you aren't using a high-pass filter

property expName

Get/set the name of the experiment or study (string)

The name will make up the first part of the EEG filename.

Example Usage:

```
rsc.expName = 'MyTestStudy'
```

property mode

Get/set the current mode.

Mode is a string that can be one of:

- 'default' or 'def' or None will exit special modes
- 'impedance' or 'imp' for impedance checking
- 'monitoring' or 'mon'
- 'test' or 'tes' to go into test view

open(expName, participant, workspace)

Opens a study/workspace on the RCS server

Parameters

- **expName** (*str*) – Name of the experiment. Will make up the first part of the EEG filename.
- **participant** (*str*) – Participant identifier. Will make up the second part of the EEG filename.
- **workspace** (*str*) – The full path to the workspace file (.rwksp), with forward slashes as path separators. e.g. "c:/myFolder/mySetup.rwksp"

openRecorder()

Opens the Recorder application from the Remote Control.

Neat, huh?!

property overwriteProtection

An attribute to get/set whether the overwrite protection is turned on.

When checking the attribute the state of *rsc.overwriteProtection* a call will be made to the RCS and the report is based on the response. There is also a variable *rsc._overwriteProtection* that is simply the stored state from the most recent call and does not make any further communication with the RCS itself.

Usage example:

```
rsc.overwriteProtection = True # set it to be on  
print(rsc.overwriteProtection) # print current state
```


property participant

Get/set the participant identifier (a string or numeric).

This identifier will make up the center part of the EEG filename.

pauseRecording ()

Pause recording EEG without ending the session.

resumeRecording ()

Resume a paused recording

sendAnnotation (annotation, annType)

Sends a message to be logged on the Recorder.

The timing of annotations may be imprecise and this should not be trusted as a method of sending sync triggers.

Annotations can contain any ASCII characters except for “;”

Parameters

- **annotation** (*string*) – The description text to be sent in the annotation.
- **annType** (*string*) – The category of the annotation which are user-defined strings (e.g. stimulus, response)
- **usage:** : (*Example*) – `rcs.sendAnnotation(“face003”, “stimulus”)`

sendRaw (message, checkOutput='OK')

A helper function to send raw messages (strings) to the RCS.

This is normally only used for debugging purposes and is not needed by most users.

Parameters

- **message** (*string*) – The string that will be sent
- **checkOutput** (*string (default='OK')*) – If a value is provided then this will be checked for by this function. If no check is needed then set `checkOutput=None`

startRecording ()

Start recording EEG.

stopRecording ()

Stop recording EEG.

property timeout

What is a reasonable timeout in seconds (initially set to 0.5)

For some systems (e.g. when the RCS is the same machine) you might want to set this to a lower value. For an unpredictable or slow network connection you might want to set this to a higher value.

property version

Reports the version of the RCS application

Example usage:

```
print(rcs.version)
```

waitForMessage (containing="", endswith="")

Wait for a message, optionally one that meets certain criteria

Parameters

- **containing** (*str*) – A string the message must contain

- **endswith** (*str*) – A string the message must end with (ignoring newline characters)

Returns

Return type The (complete) message string if one was received or None if not

waitForState (*stateName, permitted, timeout=10*)

Helper function to wait for a particular state (or any attribute, for that matter) to have a particular value. Beware this will wait indefinitely, so only call if you are confident that the state will eventually arrive!

Parameters

- **stateName** (*str*) – Name of the state (e.g. “applicationState”)
- **permitted** (*list*) – List of values that are permitted before returning

property workspace

Get/set the path to the workspace file. An absolute path is required.

Example Usage:

```
rsc.workspace = 'C:/Vision/Workfiles/testing.rwksp'
```

9.5.3 Camera

9.5.4 Overview

—
—
—
—

9.5.5 Details

9.5.6 Cedrus (response boxes)

The pyxid package, written by Cedrus, is included in the Standalone distributions.

See <https://github.com/cedrus-opensource/pyxid> for further info.

Example usage:

```
import pyxid2 as pyxid

# get a list of all attached XID devices
devices = pyxid.get_xid_devices()

dev = devices[0] # get the first device to use
if dev.is_response_device():
    dev.reset_base_timer()
    dev.reset_rt_timer()
```

(continues on next page)

(continued from previous page)

```

while True:
    dev.poll_for_response()
    if dev.response_queue_size() > 0:
        response = dev.get_next_response()
        # do something with the response
    
```

Useful functions

Device classes

9.5.7 Cambridge Research Systems Ltd.

Cambridge Research Systems makes devices to support particularly vision research.

For stimulus display

BitsPlusPlus

Control a CRS Bits# device. See typical usage in the class summary (and in the menu demos>hardware>BitsBox of PsychoPy’s Coder view).

Important: See note on *BitsPlusPlusIdentityLUT*

Attributes

<code>BitsPlusPlus(win[, contrast, gamma, ...])</code>	The main class to control a Bits++ box. This is usually a class added within the window object and is typically accessed from there. e.g.::
<code>BitsPlusPlus.setContrast(contrast[, ...])</code>	Set the contrast of the LUT for ‘bits++’ mode only :Parameters:
<code>BitsPlusPlus.setGamma(newGamma)</code>	Set the LUT to have the requested gamma value Currently also resets the LUT to be a linear contrast ramp spanning its full range.
<code>BitsPlusPlus.setLUT([newLUT, gammaCorrect, ...])</code>	Sets the LUT to a specific range of values in ‘bits++’ mode only Note that, if you leave gammaCorrect=True then any LUT values you supply will automatically be gamma corrected.

Details

```
class psychopy.hardware.crs.bits.BitsPlusPlus (win, contrast=1.0, gamma=None,
                                             nEntries=256, mode='bits++', ramp-
                                             Type='configFile', frameRate=None)
```

The main class to control a Bits++ box. This is usually a class added within the window object and is typically accessed from there. e.g.:

```
from psychopy import visual
from psychopy.hardware import crs
win = visual.Window([800,600])
bits = crs.BitsPlusPlus(win, mode='bits++')
# use bits++ to reduce the whole screen contrast by 50%:
bits.setContrast(0.5)
```

Parameters

- **contrast** – The contrast to be applied to the LUT. See `BitsPlusPlus.setLUT()` and `BitsPlusPlus.setContrast()` for flexibility on setting just a section of the LUT to a different value
- **gamma** – The value used to correct the gamma in the LUT
- **nEntries** (256) – [DEPRECATED feature]
- **mode** ('bits++' (or 'mono++' or 'color++')) – Note that, unlike the Bits#, this only affects the way the window is rendered, it does not switch the state of the Bits++ device itself (because unlike the Bits# have no way to communicate with it). The mono++ and color++ are only supported in PsychoPy 1.82.00 onwards. Even then they suffer from not having gamma correction applied on Bits++ (unlike Bits# which can apply a gamma table in the device hardware).
- **rampType** ('configFile', None or an integer) – if 'configFile' then we'll look for a valid config in the userPrefs folder if an integer then this will be used during `win.setGamma(rampType=rampType)`:
- **frameRate** (an estimate the frameRate of the monitor. If None frame rate) – will be calculated.

`_Goggles()`

(private) Used to set control the goggles. Should not be needed by user if attached to a *Window*

`_ResetClock()`

(private) Used to reset Bits hardware clock. Should not be needed by user if attached to a *Window* since this will automatically draw the reset code as part of the screen refresh.

`_drawLUTtoScreen()`

(private) Used to set the LUT in 'bits++' mode. Should not be needed by user if attached to a *Window* since this will automatically draw the LUT as part of the screen refresh.

`_drawTrigtoScreen(sendStr=None)`

(private) Used to send a trigger pulse. Should not be needed by user if attached to a *Window* since this will automatically draw the trigger code as part of the screen refresh.

`_protectTrigger()`

If Goggles (or analog) outputs are used when the digital triggers are off we need to make a set of blank triggers first. But the user might have set up triggers in waiting for a later time. So this will protect them.

`_restoreTrigger ()`

Restores the triggers to previous settings

`_setHeaders (frameRate)`

Sets up the TLock header codes and some flags that are common to operating all CRS devices

`_setupShaders ()`

creates and stores the shader programs needed for mono++ and color++ modes

`getPackets ()`

Returns the number of packets available for trigger pulses.

`primeClock ()`

Primes the clock to reset at the next screen flip - note only 1 clock reset signal will be issued but if the frame(s) after the reset frame is dropped the reset will be re-issued thus keeping timing good.

Resets continue to be issued on each video frame until the next win.flip so you need to have regular win.flips for this function to work properly.

Example:

```
bits.primeClock()
drawImage
while not response
    #do some processing
    bits.win.flip()
```

Will get a clock reset signal ready but won't issue it until the first win.flip in the loop.

`reset ()`

Deprecated: This was used on the old Bits++ to power-cycle the box. It required the compiled dll, which only worked on windows and doesn't work with Bits# or Display++.

`resetClock ()`

Issues a clock reset code using 1 screen flip if the next frame(s) is dropped the reset will be re-issued thus keeping timing good.

Resets continue to be issued on each video frame until the next win.flip so you need to have regular win.flips for this function to work properly.

Example

```
bits.resetClock() drawImage() bits.win.flip()
```

Will issue clock resets while the image is being drawn then display the image and allow the clock to continue from the same frame.

Example

```
bits.resetClock() bits.RTBoxWait() bits.win.flip()
```

Will issue clock resets until a button is pressed.

`sendTrigger (triggers=0, onTime=0, duration=0, mask=65535)`

Sends a single trigger using up 1 win.flip. The trigger will be sent on the following frame.

The triggers will continue until after the next win.flip.

Actions are always 1 frame after the request.

May do odd things if Goggles and Analog are also in use.

Example:

```
bits.sendTrigger(0b0000000010, 2.0, 4.0)
bits.win.flip()
```

Will send a 4ms pulse on DOUT1 2ms after the start of the frame. Due to the following win.flip() the pulse should last for 1 frame only.

Triggers will continue until stopTrigger is called.

setContrast (*contrast, LUTrange=1.0, gammaCorrect=None*)

Set the contrast of the LUT for 'bits++' mode only :Parameters:

contrast [float in the range 0:1] The contrast for the range being set

LUTrange [float or array] If a float is given then this is the fraction of the LUT to be used. If an array of floats is given, these will specify the start / stop points as fractions of the LUT. If an array of ints (0-255) is given these determine the start stop *indices* of the LUT

Examples

- **setContrast(1.0,0.5) to set the central 50% of the LUT so that a stimulus with** `contr=0.5` will actually be drawn with contrast 1.0
- `setContrast(1.0,[0.25,0.5])`
- **or setContrast(1.0,[63,127]) to set the lower-middle quarter of the LUT** (which might be useful in LUT animation paradigms)

setGamma (*newGamma*)

Set the LUT to have the requested gamma value Currently also resets the LUT to be a linear contrast ramp spanning its full range. May change this to read the current LUT, undo previous gamma and then apply new one?

setLUT (*newLUT=None, gammaCorrect=True, LUTrange=1.0*)

Sets the LUT to a specific range of values in 'bits++' mode only Note that, if you leave gammaCorrect=True then any LUT values you supply will automatically be gamma corrected. The LUT will take effect on the next `Window.flip()`

Examples:

- `bitsBox.setLUT()` to build a LUT using `bitsBox.contrast` and `bitsBox.gamma`
- `bitsBox.setLUT(newLUT=some256x1array)` (NB array should be float 0.0:1.0) Builds a luminance LUT using newLUT for each gun (actually array can be 256x1 or 1x256)
- `bitsBox.setLUT(newLUT=some256x3array)` (NB array should be float 0.0:1.0) Allows you to use a different LUT on each gun

(NB by using `BitsBox.setContr()` and `BitsBox.setGamma()` users may not need this function)

setTrigger (*triggers=0, onTime=0, duration=0, mask=65535*)

Quick way to set up triggers.

Triggers is a binary word that determines which triggers will be turned on.

`onTime` specifies the start time of the trigger within the frame (in S with 100uS resolution)

`Duration` specifies how long the trigger will last. (in S with 100uS resolution).

Note that `mask` only protects the digital output lines set by other activities in the Bits. Not other triggers.

Example:: `bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111) bits.startTrigger()`

Will issue a 4ms long high-going pulse, 2ms after the start of each frame on DOUT1 while protecting the value of DOUT 9.

setTriggerList (*triggerList=None, mask=65535*)

Sets up Trigger pulses in Bits++ using the fine grained method that can control every trigger line at 100uS intervals.

TriggerList should contain 1 entry for every 100uS packet (see getPackets) the binary word in each entry specifies which trigger line will be active during that time slot.

Note that mask only protects the digital output lines set by other activities in the Bits. Not other triggers.

Example:

```
packet = [0]*self._NumberPackets
packet[0] = 0b0000000010
bits.setTriggerList(packet)
```

Will sens a 100us pulse on DOUT1 at the start of the frame.

Example 2:

```
packet = [0]*self._NumberPackets
packet[10] = 0b0000000010
packet[20] = 0b0000000001
bits.setTriggerList(packet)
bits.startTrigger()
```

Will sens a 100us pulse on DOUT1 1000us after the start of the frame and a second 100us pulse on DOUT0 2000us after the start of the frame.

Triggers will continue until stopTrigger is called.

startGoggles (*left=0, right=1*)

Starts CRS stereo goggles. Note if you are using FE-1 goggles you should start this before connecting the goggles.

Left is the state of the left shutter on the first frame to be presented 0, False or 'closed'=closed; 1, True or 'open' = open,

right is the state of the right shutter on the first frame to be presented 0, False or 'closed'=closed; 1, True or 'open' = open

Note you can set the goggles to be both open or both closed on the same frame.

The system will always toggle the state of each lens so as to not damage FE-1 goggles.

Example:

```
bits.startGoggles(0,1)
bits.win.flip()
while not response:
    bits.win.flip()
    #do some processing
bits.stopGoggles()
bits.win.flip()
```

Starts toggling the goggles with the right eye open in sync with the first win.flip() within the loop. The open eye will alternate.

Example:

```
bits.startGoggles(1,1)
bits.win.flip()
while not response:
    bits.win.flip()
    #do some processing
bits.stopGoggles()
bits.win.flip()
```

Starts toggling the goggle with both eyes open in sync with the first win.flip() within the loop. Eyes will alternate between both open and both closed.

Note it is safe to leave the goggles toggling forever, ie to never call stopGoggles().

startTrigger ()

Start sending triggers on the next win flip and continue until stopped by stopTrigger Triggers start 1 frame after the frame on which the first trigger is sent.

Example:

```
bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111)
bits.startTrigger()
while imageOn:
    #do some processing
    continue
bits.stopTrigger()
bits.win.flip()
```

stopGoggles ()

Stop the stereo goggles from toggling

Example:

```
bits.startGoggles(0,1)
bits.win.flip()
while not response:
    bits.win.flip()
    #do some processing
bits.stopGoggles()
bits.win.flip()
```

Starts toggling the goggles with the right eye open in sync with the first win.flip(0) within the loop. The open eye will alternate.

Note it is safer to leave the goggles toggling forever, ie to never call stopGoggles().

stopTrigger ()

Stop sending triggers at the next win flip

Example:

```
bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111)
bits.startTrigger()
while imageOn:
    #do some processing
    continue
bits.stopTrigger()
bits.win.flip()
```

syncClocks (t)

Synchronise the Bits/RTBox Clock with the host clock Given by t.

Finding the identity LUT

For the Bits++ (and related) devices to work correctly it is essential that the graphics card is not altering in any way the values being passed to the monitor (e.g. by gamma correcting). It turns out that finding the ‘identity’ LUT, where exactly the same values come out as were put in, is not trivial. The obvious LUT would have something like 0/255, 1/255, 2/255... in entry locations 0,1,2... but unfortunately most graphics cards on most operating systems are ‘broken’ in one way or another, with rounding errors and incorrect start points etc.

provides a few of the common variants of LUT and that can be chosen when you initialise the device using the parameter *rampType*. If no *rampType* is specified then will choose one for you:

```
from psychopy import visual
from psychopy.hardware import crs

win = visual.Window([1024,768], useFBO=True) #we need to be rendering to_
↳framebuffer
bits = crs.BitsPlusPlus(win, mode = 'bits++', rampType = 1)
```

The Bits# is capable of reporting back the pixels in a line and this can be used to test that a particular LUT is indeed providing identity values. If you have previously connected a *BitsSharp* device and used it with then a file will have been stored with a LUT that has been tested with that device. In this case set *rampType* = “*configFile*” for PsychoPy to use it if such a file is found.

BitsSharp

Control a CRS Bits# device. See typical usage in the class summary (and in the menu demos>hardware>BitsBox of PsychoPy’s Coder view).

Attributes

<i>BitsSharp</i> ([win, portName, mode, ...])	A class to support functions of the Bits# (and most Display++ functions This device uses the CDC (serial port) connection to the Bits box.
<i>BitsSharp.mode</i>	Get/set the mode of the BitsSharp to one of: “bits++” “mono++” “color++” “status” “storage” “auto”
<i>BitsSharp.isAwake</i> ()	Test whether we have an active connection on the virtual serial port
<i>BitsSharp.getInfo</i> ()	Returns a python dictionary of info about the Bits Sharp box
<i>BitsSharp.checkConfig</i> ([level, demoMode, logFile])	Checks whether there is a configuration for this device and whether it’s correct
<i>BitsSharp.gammaCorrectFile</i>	Get / set the gamma correction file to be used (as stored on the device)
<i>BitsSharp.temporalDithering</i>	Temporal dithering can be set to True or False
<i>BitsSharp.monitorEDID</i>	Get / set the EDID file for the monitor.
<i>BitsSharp.beep</i> ([freq, dur])	Make a beep of a given frequency and duration
<i>BitsSharp.getVideoLine</i> (lineN, nPixels[, ...])	Return the r,g,b values for a number of pixels on a particular video line
<i>BitsSharp.start</i> ()	[Not currently implemented] Used to begin event collection by the device.

continues on next page

Table 9.40 – continued from previous page

<code>BitsSharp.stop()</code>	[Not currently implemented] Used to stop event collection by the device.
-------------------------------	--

Direct communications with the serial port:

<code>BitsSharp.sendMessage(message[, autoLog])</code>	Send a command to the device (does not wait for a reply or sleep())
<code>BitsSharp.getResponse([length, timeout])</code>	Read the latest response from the serial port

Control the CLUT (Bits++ mode only):

<code>BitsSharp.setContrast(contrast[, LUTrange, ...])</code>	Set the contrast of the LUT for ‘bits++’ mode only :Parameters:
<code>BitsSharp.setGamma(newGamma)</code>	Set the LUT to have the requested gamma value Currently also resets the LUT to be a linear contrast ramp spanning its full range.
<code>BitsSharp.setLUT([newLUT, gammaCorrect, ...])</code>	SetLUT is only really needed for bits++ mode of bits# to set the look-up table (256 values with 14bits each).

Details

class psychopy.hardware.crs.bits.**BitsSharp** (*win=None, portName=None, mode="", checkConfigLevel=1, gammaCorrect='hardware', gamma=None, noComms=False*)

A class to support functions of the Bits# (and most Display++ functions) This device uses the CDC (serial port) connection to the Bits box. To use it you must have followed the instructions from CRS Ltd. to get your box into the CDC communication mode. Typical usage (also see demo in Coder view demos>hardware>BitsBox):

```
from psychopy import visual
from psychopy.hardware import crs
# we need to be rendering to framebuffer
win = visual.Window([1024,768], useFBO=True)
bits = crs.BitsSharp(win, mode = 'mono++')
# You can continue using your window as normal and OpenGL shaders
# will convert the output as needed
print(bits.info)
if not bits.OK:
    print('failed to connect to Bits box')
    core.quit()
core.wait(0.1)
# now, you can change modes using
bits.mode = 'mono++' # 'color++', 'mono++', 'bits++', 'status'
```

Note that the firmware in Bits# boxes varies over time and some features of this class may not work for all firmware versions. Also Bits# boxes can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then ‘status’ and ‘event’ commands in this class may not work.

RTBox commands that reset the key mapping have been found not to work on some firmware

Parameters

- **win** (a PsychoPy *Window* object, required) –
- **portName** (*str or int*) – the (virtual) serial port to which the device is connected. If None then PsychoPy will search available serial ports and test communication (on OSX, the first match of */dev/tty.usbmodemfa** will be used and on linux */dev/ttyS0* will be used)
- **mode** (*'bits++', 'color++', 'mono++', 'status'*) –
- **checkConfigLevel** (*int*) – Allows you to specify how much checking of the device is done to ensure a valid identity look-up table. If you specify one level and it fails then the check will be escalated to the next level (e.g. if we check level 1 and find that it fails we try to find a new LUT):
 - 0 don't check at all
 - **1 check that the graphics driver and OS version haven't** changed since last LUT calibration
 - **2 check that the current LUT calibration still provides** identity (requires switch to status mode)
 - **3 search for a new identity look-up table (requires** switch to status mode)
- **gammaCorrect** (*string*) – Governing how gamma correction is performed: - 'hardware': use the gamma correction file stored on the hardware
 - 'FBO': gamma correct using shaders when rendering the FBO to back buffer
 - 'bitsMode': in bits++ mode there is a user-controlled LUT that we can use for gamma correction
- **noComms** (*bool*) – If True then don't try to communicate with the device at all (passive mode). This can be useful if you want to debug the system without actually having a Bits# connected.

RTBoxAddKeys (*map*)

Add key mappings to an existing map. RTBox events can be mapped to a number of physical events on Bits# They can be mapped to digital input lines, triggers and CB6 IR input channels. The format for map is a list of tuples with each tuple containing the name of the RTBox button to be mapped and its source eg ('btn1','Din1') maps physical input Din1 to logical button btn1. RTBox has four logical buttons (btn1-4) and three auxiliary events (light, pulse and trigger) Buttons/events can be mapped to multiple physical inputs and stay mapped until reset.

Example:

```
bits.RTBoxSetKeys([('btn1', 'Din0'), ('btn2', 'Din1')])
bits.RTBoxAddKeys([('btn1', 'IRButtonA'), ('btn2', 'IRButtonB')])
```

Will link Din0 to button 1 and Din1 to button 2. Then adds IRButtonA and IRButtonB alongside the original mappings.

Now both hard wired and IR inputs will - emulating the same logical button press.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

RTBoxCalibrate (*N=1*)

Used to assess error between host clock and Bits# button press time stamps.

Prints each sample provided and returns the mean error.

The clock will never be completely in sync but the aim is that there should be that the difference between them should not grow over a series of button presses.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

RTBoxClear ()

Flushes the serial input buffer. Its good to do this before and after data collection. This just calls flush() so is a wrapper for RTBox.

RTBoxDisable ()

Disables the detection of RTBox events. This is useful to stop the Bits# from reporting key presses When you no longer need them. Nad must be done before using any other data logging methods.

It undoes any button - input mappings.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

The ability to reset keys mappings has been found not to work on some Bits# firmware.

RTBoxEnable (*mode=None, map=None*)

Sets up the RTBox with preset or bespoke mappings and enables event detection.

RTBox events can be mapped to a number of physical events on Bits# They can be mapped to digital input lines, tigers and CB6 IR input channels.

Mode is a list of strings. Preset mappings provided via mode:

- *CB6* for the CRS CB6 IR response box.
- *IO* for a three button box connected to Din0-2
- *IO6* for a six button box connected to Din0-5

If mode = None or is not set then the value of self.RTBoxMode is used.

Bespoke Mappings over write preset ones.

The format for map is a list of tuples with each tuple containing the name of the RT Box button to be mapped and its source eg ('btn1','Din0') maps physical input Din0 to logical button btn1.

Note the lowest number button event is Btn1

RTBox has four logical buttons (btn1-4) and three auxiliary events (light, pulse and trigger) Buttons/events can be mapped to multiple physical inputs and stay mapped until reset.

Mode is a list of string or list of strings that contains keywords to determine present mappings and modes for RTBox.

- If mode includes 'Down' button events will be detected when pressed.
- If mode includes 'Up' button events will be detected when released.

You can detect both types of event but note that pulse, light and trigger events don't have an 'Up' mode.

If Trigger is included in mode the trigger event will be mapped to the trigIn connector.

Example: .. code-block:: python

```
bits.RTBoxEnable(mode = ['Down'], map = [('btn1','Din0'), ('btn2','Din1')])
```

enables the RTBox emulation to detect Down events on buttons 1 and 2 where they are mapped to DIN0 and DIN1.

Example: .. code-block:: python

```
bits.RTBoxEnable(mode = ['Down','CB6'])
```

enables the RTBox emulation to detect Down events on the standard CB6 IR response box keys.

If no key direction has been set (mode does not contain 'Up' or 'Down') the default is 'Down'.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

The ability to reset keys mappings has been found not to work on some Bits# firmware.

RTBoxKeysPressed (*N=1*)

Check to see if (at least) the appropriate number of RTBox style key presses have been made.

Example

```
bits.RTBoxKeysPressed(5)
```

will return false until 5 button presses have been recorded.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

RTBoxResetKeys ()

Resets the key mappings to no mapping. Has the effect of disabling RTBox input.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

The ability to reset keys mappings has been found not to work on some Bits# firmware.

RTBoxSetKeys (*map*)

Set key mappings: first resets existing then adds new ones. Does not reset any event that is not in the new list. RTBox events can be mapped to a number of physical events on Bits# They can be mapped to digital input lines, triggers and CB6 IR input channels. The format for map is a list of tuples with each tuple containing the name of the RTBox button to be mapped and its source eg ('btn1','Din1') maps physical input Din1 to logical button btn1.

RTBox has four logical buttons (btn1-4) and three auxiliary events (light, pulse and trigger) Buttons/events can be mapped to multiple physical inputs and stay mapped until reset.

Example

```
bits.RTBoxSetKeys([('btn1','Din0'),('light','Din9']])
```

Will link Din0 to button 1 and Din9 to the the light input emulation.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

RTBoxWait ()

Waits until (at least) one of RTBox style key presses have been made Pauses program execution in mean time.

Example

```
res = bits.RTBoxWait()
```

will suspend all other activity until 1 button press has been recorded and will then return a dict / structure containing results.

Results can be accessed as follows:

structure res.dir, res.button, res.time

or dictionary res['dir'], res['button'], res['time']

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

RTBoxWaitN (N=1)

Waits until (at least) the appropriate number of RTBox style key presses have been made Pauses program execution in mean time.

Example

```
res = bits.RTBoxWaitN(5)
```

will suspend all other activity until 5 button presses have been recorded and will then return a list of Dicts containing the 5 results.

Results can be accessed as follows:

structure res[0].dir, res[0].button, res[0].time

or dictionary res[0]['dir'], res[0]['button'], res[0]['time']

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

__Goggles ()

(private) Used to set control the goggles. Should not be needed by user if attached to a *Window*

`__RTBoxDecodeResponse` (*msg, N=1*)

Helper function for decoding key presses in the RT response box format.

Not normally needed by user

`__ResetClock` ()

(private) Used to reset Bits hardware clock. Should not be needed by user if attached to a *Window* since this will automatically draw the reset code as part of the screen refresh.

`__drawLUTtoScreen` ()

(private) Used to set the LUT in 'bits++' mode. Should not be needed by user if attached to a *Window* since this will automatically draw the LUT as part of the screen refresh.

`__drawTrigtoScreen` (*sendStr=None*)

(private) Used to send a trigger pulse. Should not be needed by user if attached to a *Window* since this will automatically draw the trigger code as part of the screen refresh.

`__extractStatusEvents` ()

Interprets values from status log to pullout any events.

Should not be needed by user if start/stopStatusLog or pollStatus are used

Fills statusEvents with a list of dictionary like objects with the following entries source, input, direction, time.

source = the general source of the event - e.g. DIN for Digital input, IR for IT response box

input = the individual input in the source. direction = 'up' or 'down' time = time stamp.

Events are recorded relative to the four event flags statusDINBase, initial values for digital ins. statusIRBase, initial values for CB6 IR box. statusTrigInBase, initial values for TrigIn. statusMode, direction(s) of events to be reported.

The data can be accessed as statusEvents[i]['time'] or statusEvents[i].time

Also set status._nEvents to the number of events recorded

`__getStatusLog` ()

Read the log Queue

Should not be needed by user if start/stopStatusLog or pollStatus are used.

fills statusValues with a list of dictionary like objects with the following entries: sample, time, trigIn, DIN[10], DWORD, IR[6], ADC[6]

They can be accessed as statusValues[i]['sample'] or statusValues[i].sample, statusValues[i].ADC[j]

Also sets status._nValues to the number of values recorded.

`__inWaiting` ()

Helper function to determine how many bytes are waiting on the serial port.

`__protectTrigger` ()

If Goggles (or analog) outputs are used when the digital triggers are off we need to make a set of blank triggers first. But the user might have set up triggers in waiting for a later time. So this will protect them.

`__restoreTrigger` ()

Restores the triggers to previous settings

`__setHeaders` (*frameRate*)

Sets up the TLock header codes and some flags that are common to operating all CRS devices

`__setupShaders` ()

creates and stores the shader programs needed for mono++ and color++ modes

`_statusBox()`

Should not normally be called by user Called in its own thread via `self.statusBoxEnable()` Reads the status reports from the Bits# for default 60 seconds or until `self.statusBoxDisable()` is called.

Note any non status reports are found on the buffer will cause an error.

`args` specifies the time over which to record status events. The minimum time is 10ms, less than this results in recording stopping after about 1 status report has been read.

Puts its results into a Queue.

This function is normally run in its own thread so actions can be asynchronous.

`_statusDisable()`

Stop Bits# from recording data - and clears the buffer

Not normally needed by user

`_statusEnable()`

Sets the Bits# to continuously send back its status until stopped. You get a lot a data by leaving this going.

Not normally needed by user

`_statusLog(args=60)`

Should not normally be called by user Called in its own thread via `self.startStatusLog()` Reads the status reports from the Bits# for default 60 seconds or until `self.stopStatusLog()` is called. Ignores the last line as this is can be bogus. Note any non status reports are found on the buffer will cause an error.

`args` specifies the time over which to record status events. The minimum time is 10ms, less than this results in recording stopping after about 1 status report has been read.

Puts its results into a Queue.

This function is normally run in its own thread so actions can be asynchronous.

`beep(freq=800, dur=1)`

Make a beep of a given frequency and duration

`checkConfig(level=1, demoMode=False, logFile="")`

Checks whether there is a configuration for this device and whether it's correct

Parameters `level` (*integer*) –

- 0: do nothing
- **1: check that we have a config file and that the graphics** card and operating system match that specified in the file. Then assume identity LUT is correct
- **2: switch the box to status mode and check that the** identity LUT is currently working
- 3: force a fresh search for the identity LUT

`clock()`

Reads the internal clock of the Bits box via the RTBox format but note there will be a delay in reading the value back. The format for the return values is the same as for button box presses. The return value for button will be 9 and the return value for event will be time. The return value for time will be the time of the clock at the moment of the request.

Example

```
res = bits.clock() print(res.time) print(res['time'])
```

driverFor = []

flush ()

Flushes the serial input buffer Its good to do this before and after data collection, And generally quite often.

property gammaCorrectFile

Get / set the gamma correction file to be used (as stored on the device)

getAllRTBoxResponses ()

Read all of the RTBox style key presses on the input buffer. Returns a list of dict like objects with three members 'button', 'dir' and 'time'

'button' is a number from 1 to 9 to indicate the event that was detected. 1-4 are the 'btn1-btn4' events, 5 and 6 are the 'light' and 'pulse' events, 7 is the 'trigger' event, 9 is a requested timestamp event (see Clock()).

'dir' is the direction of the event eg 'up' or 'down', trigger is described as 'on' when low.

'dir' is set to 'time' if a requested timestamp event has been detected.

'time' is the timestamp associated with the event.

Values can be read as a structure eg:

```
res = getAllRTBoxResponses()
res[0].dir, res[0].button, res[0].time
```

or dictionary:

```
res[0]['dir'], res[0]['button'], res[0]['time']
```

Note even if only 1 key press was found a list of dict / objects is returned

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

getAllStatusBoxResponses ()

Read all of the statusBox style key presses on the input buffer. Returns a list of dict like objects with three members 'button', 'dir' and 'time'

'button' is a number from 1 to 9 to indicate the event that was detected. 1-17 are the 'btn1-btn17' events.

'dir' is the direction of the event eg 'up' or 'down', trigger is described as 'on' when low.

'dir' is set to 'time' if a requested timestamp event has been detected.

'time' is the timestamp associated with the event.

Values can be read as a structure eg:

```
res= getAllStatusBoxResponses()
res[0].dir, res[0].button, res[0].time
```

or dictionary:

```
res[0]['dir'], res[0]['button'], res[0]['time']
```

Note even if only 1 key press was found a list of dict / objects is returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getAllStatusEvents ()

Returns the whole status event list

Returns a list of dictionary like objects with the following entries source, input, direction, time.

source = the general source of the event - e.g. DIN for Digital input, IR for CB6 IR response box events

input = the individual input in the source. direction = 'up' or 'down' time = time stamp.

All sources are numbered from zero. Din 0 ... 9 IR 0 ... 5 ADC 0 ... 5

mode specifies which directions of events are captured. e.g 'up' will only report up events.

The data can be accessed as value[i]['time'] or value[i].time

Example:

```
bits.startStatusLog()
while not event
    #do some processing
    continue
bits.stopStatusLog()
res=getAllStatusEvents()
print (bits.res[0].time)
```

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getAllStatusValues ()

Returns the whole status values list.

Returns a list of dict like objects with the following entries sample, time, trigIn, DIN[10], DWORD, IR[6], ADC[6] sample is the sample ID number. time is the time stamp. trigIn is the value of the trigger input. DIN is a list of 10 digital input values. DWORD represents the digital inputs as a single decimal value. IR is a list of 10 infra-red (IR) input values. ADC is a list of 6 analog input values. These can be accessed as value[i]['sample'] or value[i].sample, values[i].ADC[j].

All sources are numbered from zero. Din 0 ... 9 IR 0 ... 5 ADC 0 ... 5

Example:

```
bits.startStatusLog()
while not event
    #do some processing
    continue
bits.stopStatusLog()
res=getAllStatusValues()
print (bits.res[0].time)
```

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getAnalog ($N=0$)

Pulls out the values of the analog inputs for the Nth status entry.

Returns a dictionary with a list of 6 floats (ADC) and a time stamp (time).

All sources are numbered from zero. ADC 0 ... 5

Example

```
bits.pollStatus() res=bits.getAnalog() print(res['ADC'])
```

will poll the status display the values of the ADC inputs in the first status entry returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getDigital ($N=0$)

Pulls out the values of the digital inputs for the Nth status entry.

Returns a dictionary with a list of 10 ints that are 1 or 0 (DIN) and a time stamp (time)

Il sources are numbered from zero. Din 0 ... 9

Example

```
bits.pollStatus() res=bits.getAnalog() print(res['DIN'])
```

will poll the status display the value of the digital inputs in the first status entry returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also DBits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getDigitalWord ($N=0$)

Pulls out the values of the digital inputs for the Nth status entry.

Returns a dictionary with a 10 bit word representing the binary values of those inputs (DWORD) and a time stamp (time).

Example

```
bits.pollStatus() res=bits.getAnalog() print(res['DWORD'])
```

will poll the status display the value of the digital inputs as a decimal number.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getIRBox (*N=0*)

Pulls out the values of the CB6 IR response box inputs for the Nth status entry.

Returns a dictionary with a list of 6 ints that are 1 or 0 (IRBox) and a time stamp (time).

Il sources are numbered from zero. IR 0 ... 5

Example

```
bits.pollStatus() res=bits.getAnalog() print(res['IRBox'])
```

will poll the status display the values of the IR box buttons in the first status entry returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getInfo ()

Returns a python dictionary of info about the Bits Sharp box

Example:: info=bits.getInfo print(info['ProductType'])

getPackets ()

Returns the number of packets available for trigger pulses.

getRTBoxResponse ()

checks for one RTBox style key presses on the input buffer then reads it. Returns a dict like object with three members 'button', 'dir' and 'time'

'button' is a number from 1 to 9 to indicate the event that was detected. 1-4 are the 'btn1-btn4' events, 5 and 6 are the 'light' and 'pulse' events, 7 is the 'trigger' event, 9 is a requested timestamp event (see Clock()).

'dir' is the direction of the event eg 'up' or 'down', trigger is described as 'on' when low.

'dir' is set to 'time' if a requested timestamp event has been detected.

'time' is the timestamp associated with the event.

Value can be read as a structure, eg: res= getRTBoxResponse() res.dir, res.button, res.time

or dictionary res['dir'], res['button'], res['time']

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

getRTBoxResponses ($N=1$)

checks for (at least) an appropriate number of RTBox style key presses on the input buffer then reads them. Returns a list of dict like objects with three members 'button', 'dir' and 'time'

'button' is a number from 1 to 9 to indicate the event that was detected. 1-4 are the 'btn1-btn4' events, 5 and 6 are the 'light' and 'pulse' events, 7 is the 'trigger' event, 9 is a requested timestamp event (see Clock()).

'dir' is the direction of the event eg 'up' or 'down', trigger is described as 'on' when low.

'dir' is set to 'time' if a requested timestamp event has been detected.

'time' is the timestamp associated with the event.

Values can be read as a list of structures eg:

```
res = getRTBoxResponses(3)
res[0].dir, res[0].button, res[0].time
```

or dictionaries:

```
res[0]['dir'], res[0]['button'], res[0]['time']
```

Note even if only 1 key press was requested a list of dict / objects is returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. Such variations may affect key mappings for RTBox commands.

getResponse ($length=1, timeout=0.1$)

Read the latest response from the serial port

Parameters:

length determines whether we expect:

- 1: a single-line reply (use readline())
- 2: a multiline reply (use readlines() which *requires* timeout)
- **-1: may not be any EOL character; just read whatever chars are there**

getStatus ($N=0$)

Pulls out the Nth entry in the statusValues list.

Returns a dict like object with the following entries sample, time, trigIn, DIN[10], DWORD, IR[6], ADC[6]

sample is the sample ID number. time is the time stamp. trigIn is the value of the trigger input. DIN is a list of 10 digital input values. DWORD represents the digital inputs as a single decimal value. IR is a list of 10 infra-red (IR) input values. ADC is a list of 6 analog input values. These can be accessed as value['sample'] or value.sample, values.ADC[j].

All sources are numbered from zero. Din 0 ... 9 IR 0 ... 5 ADC 0 ... 5

Example:

```
bits.startStatusLog()
while not event
    #do some processing
    continue
bits.stopStatusLog()
```

(continues on next page)

(continued from previous page)

```
res=getStatus(20)
print(bits.res.time)
```

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getStatusBoxResponse()

checks for one statusBox style key presses on the input buffer then reads it. Returns a dict like object with three members 'button', 'dir' and 'time'

'button' is a number from 1 to 9 to indicate the event that was detected. 1-17 are the 'btn1-btn17' events.

'dir' is the direction of the event eg 'up' or 'down', trigger is described as 'on' when low.

'dir' is set to 'time' if a requested timestamp event has been detected.

'time' is the timestamp associated with the event.

Value can be read as a structure, eg: res= getRTBoxResponse() res.dir, res.button, res.time

or dictionary res['dir'], res['button'], res['time']

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getStatusBoxResponses(N=1)

checks for (at least) an appropriate number of RTBox style key presses on the input buffer then reads them. Returns a list of dict like objects with three members 'button', 'dir' and 'time'

'button' is a number from 1 to 9 to indicate the event that was detected. 1-4 are the 'btn1-btn4' events, 5 and 6 are the 'light' and 'pulse' events, 7 is the 'trigger' event, 9 is a requested timestamp event (see Clock()).

'dir' is the direction of the event eg 'up' or 'down', trigger is described as 'on' when low.

'dir' is set to 'time' if a requested timestamp event has been detected.

'time' is the timestamp associated with the event.

Values can be read as a list of structures eg:

```
res = getRTBoxResponses(3)
print(res[0].dir, res[0].button, res[0].time)
```

or dictionaries:

```
print(res[0]['dir'], res[0]['button'], res[0]['time'])
```

Note even if only 1 key press was requested a list of dict / objects is returned.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getStatusEvent (*N=0*)

pulls out the Nth event from the status event list

Returns a dictionary like object with the following entries source, input, direction, time.

source = the general source of the event - e.g. DIN for Digital input, IR for IT response box.

input = the individual input in the source. direction = 'up' or 'down' time = time stamp.

All sources are numbered from zero. Din 0 ... 9 IR 0 ... 5 ADC 0 ... 5

mode specifies which directions of events are captured, e.g 'up' will only report up events.

The data can be accessed as value['time'] or value.time

Example:

```
bits.startStatusLog()
while not event
    #do some processing
    continue
bits.stopStatusLog()
res=getAllStatusEvents(20)
print(bits.res.time)
```

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getTrigIn (*N=0*)

Pulls out the values of the trigger input for the Nth status entry.

Returns dictionary with a 0 or 1 (trigIn) and a time stamp (time)

Example

```
bits.pollStatus() res=bits.getAnalog() print(res['trigIn'])
```

will poll the status display the value of the trigger input.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

getVideoLine (*lineN, nPixels, timeout=10.0, nAttempts=10*)

Return the r,g,b values for a number of pixels on a particular video line

Parameters

- **lineN** – the line number you want to read
- **nPixels** – the number of pixels you want to read
- **nAttempts** – the first time you call this function it has to get to status mode. In this case it sometimes takes a few attempts to make the call work

Returns an Nx3 numpy array of uint8 values

isAwake ()

Test whether we have an active connection on the virtual serial port

property isOpen

longName = ''

property mode

Get/set the mode of the BitsSharp to one of: “bits++” “mono++” “color++” “status” “storage” “auto”

property monitorEDID

Get / set the EDID file for the monitor. The edid files will be located in the EDID subdirectory of the flash disk. The file *automatic.edid* will be the file read from the connected monitor.

name = b'CRS Bits#'

pause ()

Pause for a default period for this device

pollStatus (t=0.0001)

Reads the status reports from the Bits# for the specified usually short time period t. The script will wait for this time to lapse so not ideal for time critical applications.

If t is less than 0.01 polling will continue until at least 1 data entry has been recorded.

If you don't want to wait while this does its job use `startStatusLog` and `stopStatusLog` instead.

Fills the `statusValues` list with all the status values read during the time period.

Fills the `statusEvents` list with just those status values that are likely to be meaningful events.

the members `statusValues` and `statusEvents` will end up containing dict like objects of the following style: `sample, time, trigIn, DIN[10], DWORD, IR[6], ADC[6]`

They can be accessed as `statusValues[i]['sample']` or `statusValues[i].sample`, `statusValues[x].ADC[j]`.

Example:

```
bits.pollStatus()
print(bits.statusValues[0].IR[0])
```

will display the value of the IR InputA in the first sample recorded.

Note: Starts and stops logging for itself.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their `config.xml` file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your `config.xml` file then ‘status’ and ‘event’ commands in this class may not work.

primeClock ()

Primes the clock to reset at the next screen flip - note only 1 clock reset signal will be issued but if the frame(s) after the reset frame is dropped the reset will be re-issued thus keeping timing good.

Resets continue to be issued on each video frame until the next `win.flip` so you need to have regular `win.flips` for this function to work properly.

Example:

```
bits.primeClock()
drawImage
while not response
```

(continues on next page)

(continued from previous page)

```
#do some processing
bits.win.flip()
```

Will get a clock reset signal ready but won't issue it until the first win.flip in the loop.

read (*timeout=0.1*)

Get the current waiting characters from the serial port if there are any.

Mostly used internally but may be needed by user. Note the return message depends on what state the device is in and will need to be decoded. See the Bits# manual but also the other functions herein that do the decoding for you.

Example

```
message = bits.read()
```

reset ()

Deprecated: This was used on the old Bits++ to power-cycle the box. It required the compiled dll, which only worked on windows and doesn't work with Bits# or Display++.

resetClock ()

Issues a clock reset code using 1 screen flip if the next frame(s) is dropped the reset will be re-issued thus keeping timing good.

Resets continue to be issued on each video frame until the next win.flip so you need to have regular win.flips for this function to work properly.

Example

```
bits.resetClock() drawImage() bits.win.flip()
```

Will issue clock resets while the image is being drawn then display the image and allow the clock to continue from the same frame.

Example

```
bits.resetClock() bits.RTBoxWait() bits.win.flip()
```

Will issue clock resets until a button is pressed.

sendAnalog (*AOUT1=0, AOUT2=0*)

sends a single analog output pulse uses up 1 win flip. pulse will continue until next win flip called. Actions are always 1 frame behind the request.

May conflict with trigger and goggle settings.

Example

```
bits.sendAnalog(4.5,-2.0) bits.win.flip()
```

sendMessage (*message, autoLog=True*)

Send a command to the device (does not wait for a reply or sleep())

sendTrigger (*triggers=0, onTime=0, duration=0, mask=65535*)

Sends a single trigger using up 1 win.flip. The trigger will be sent on the following frame.

The triggers will continue until after the next win.flip.

Actions are always 1 frame after the request.

May do odd things if Goggles and Analog are also in use.

Example:

```
bits.sendTrigger(0b0000000010, 2.0, 4.0)
bits.win.flip()
```

Will send a 4ms pulse on DOUT1 2ms after the start of the frame. Due to the following win.flip() the pulse should last for 1 frame only.

Triggers will continue until stopTrigger is called.

setAnalog (*AOUT1=0, AOUT2=0*)

Sets up Analog outputs in Bits# AOUT1 and AOUT2 are the two analog values required in volts. Analog commands are issued at the next win.flip() and actioned 1 video frame later.

Example

```
bits.set Analog(4.5,-2.2) bits.startAnalog() bits.win.flip()
```

setContrast (*contrast, LUTrange=1.0, gammaCorrect=None*)

Set the contrast of the LUT for 'bits++' mode only :Parameters:

contrast [float in the range 0:1] The contrast for the range being set

LUTrange [float or array] If a float is given then this is the fraction of the LUT to be used. If an array of floats is given, these will specify the start / stop points as fractions of the LUT. If an array of ints (0-255) is given these determine the start stop *indices* of the LUT

Examples

- **setContrast(1.0,0.5)** to set the central 50% of the LUT so that a stimulus with `contr=0.5` will actually be drawn with contrast 1.0
- **setContrast(1.0,[0.25,0.5])**
- or **setContrast(1.0,[63,127])** to set the lower-middle quarter of the LUT (which might be useful in LUT animation paradigms)

setGamma (*newGamma*)

Set the LUT to have the requested gamma value Currently also resets the LUT to be a linear contrast ramp spanning its full range. May change this to read the current LUT, undo previous gamma and then apply new one?

setLUT (*newLUT=None, gammaCorrect=False, LUTrange=1.0, contrast=None*)

SetLUT is only really needed for bits++ mode of bits# to set the look-up table (256 values with 14bits each). For the BitsPlusPlus device the default is to perform gamma correction here but on the BitsSharp it seems better to have the device perform that itself as the last step so gamma correction is off here by default. If no contrast has yet been set (it isn't needed for other modes) then it will be set to 1 here.

setRTBoxMode (*mode=['CB6', 'Down', 'Trigger']*)

Sets the RTBox mode data member - does not actually set the RTBox into this mode.

Example

```
bits.setRTBoxMode(['CB6','Down']) # set the mode
bits.RTBoxEnable() # Enable RTBox emulation with # the preset mode.
```

sets the RTBox mode settings for a CRS CB6 button box. and for detection of 'Down' events only.

setStatusBoxMode (*mode=['CB6', 'Down', 'Trigger', 'Analog']*)

Sets the statusBox mode data member - does not actually set the statusBox into this mode.

Example

```
bits.setStatusBoxMode(['CB6','Down']) # set the mode
bits.statusBoxEnable() # Enable status Box emulation with # the preset mode.
```

sets the statusBox mode settings for a CRS CB6 button box. and for detection of 'Down' events only.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

setStatusBoxThreshold (*threshold=None*)

Sets the threshold by which analog inputs must change to trigger a button press event. If None the threshold will be set very high so that no such events are triggered.

Can be used to change the threshold for analog events without having to re enable the status box system as a whole.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

setStatusEventParams (*DINBase=1023, IRBase=63, TrigInBase=0, ADCBase=0, threshold=9999.99, mode=['up', 'down']*)

Sets the parameters used to determine if a status value represents a reportable event.

DIN_base = a 10 bit binary word specifying the expected starting values of the 10 digital input lines

IR_base = a 6 bit binary word specifying the expected starting values of the 6 CB6 IR buttons

Trig_base = the starting value of the Trigger input

mode = a list of event types to monitor can be 'up' or 'down' typically 'down' corresponds to a button press or when the input is being pulled down to zero volts.

Example:

```

bits.setStatusEventParams (DINBase=0b1111111111,
                          IRBase=0b1111111,
                          TrigInBase=0,
                          ADCBase=0,
                          threshold = 3.4,
                          mode = ['down'])

bits.startStatusLog()
while not event
    #do some processing
    continue
bits.stopStatusLog()
res=getAllStatusEvents(0)
print(bits.res.time)

```

This will start the event extraction process as if DINs and IRs are all '1', Trigger is '0' ADCs = 0 with an ADC threshold for change of 3.4 volts, and will only register 'down' events. Here we display the time stamp of the first event.

Note that the firmware in Display++ units varies over time and some features of this class may not work for all firmware versions. Also Display++ units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these reports are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

setTrigger (*triggers=0, onTime=0, duration=0, mask=65535*)

Quick way to set up triggers.

Triggers is a binary word that determines which triggers will be turned on.

onTime specifies the start time of the trigger within the frame (in S with 100uS resolution)

Duration specifies how long the trigger will last. (in S with 100uS resolution).

Note that mask only protects the digital output lines set by other activities in the Bits. Not other triggers.

Example:

```

` bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111) bits.
startTrigger() `

```

Will issue a 4ms long high-going pulse, 2ms after the start of each frame on DOUT1 while protecting the value of DOUT 9.

setTriggerList (*triggerList=None, mask=65535*)

Overload of Bits# and Display++ Sets up Trigger pulses via the list method while preserving the analog output settings.

Sets up Trigger pulses in Bist++ using the fine grained method that can control every trigger line at 100uS intervals.

TriggerList should contain 1 entry for every 100uS packet (see getPackets) the binary word in each entry specifies which trigger line will be active during that time slot.

Note that mask only protects the digital output lines set by other activities in the Bits. Not other triggers.

Example

```
packet = [0]*self._NumberPackets packet[0] = 0b0000000010 bits.setTriggerList(packet)
```

Will sens a 100us pulse on DOUT1 at the start of the frame.

Example 2: `packet = [0]*self._NumberPackets packet[10] = 0b0000000010 packet[20] = 0b0000000001 bits.setTriggerList(packet) bits.startTrigger()`

Will sens a 100us pulse on DOUT1 1000us after the start of the frame and a second 100us pulse on DOUT0 2000us after the start of the frame.

Triggers will continue until stopTrigger is called.

start ()

[Not currently implemented] Used to begin event collection by the device.

Not really needed as other members now do this.

startAnalog ()

will start sending analog signals on the next win flip and continue until stopped.

Example

```
bits.set Analog(4.5,-2.2) bits.startAnalog() bits.win.flip()
```

startGoggles (left=0, right=1)

Starts CRS stereo goggles. Note if you are using FE-1 goggles you should start this before connecting the goggles.

Left is the state of the left shutter on the first frame to be presented 0, False or 'closed'=closed; 1, True or 'open' = open,

right is the state of the right shutter on the first frame to be presented 0, False or 'closed'=closed; 1, True or 'open' = open

Note you can set the goggles to be both open or both closed on the same frame.

The system will always toggle the state of each lens so as to not damage FE-1 goggles.

Example:

```
bits.startGoggles(0,1)
bits.win.flip()
while not response:
    bits.win.flip()
    #do some processing
bits.stopGoggles()
bits.win.flip()
```

Starts toggling the goggles with the right eye open in sync with the first win.flip() within the loop. The open eye will alternate.

Example:

```
bits.startGoggles(1,1)
bits.win.flip()
while not response:
    bits.win.flip()
    #do some processing
```

(continues on next page)

(continued from previous page)

```
bits.stopGoggles()
bits.win.flip()
```

Starts toggling the goggle with both eyes open in sync with the first win.flip() within the loop. Eyes will alternate between both open and both closed.

Note it is safe to leave the goggles toggling forever, ie to never call stopGoggles().

startStatusLog (*t=60*)

Start logging data from the Bits#

Starts data logging in its own thread.

Will run for t seconds, default 60 or until stopStatusLog() is called.

Example:

```
bits.startStatusLog()
while not event
    #do some processing
    continue
bits.stopStatusLog()
```

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

startTrigger ()

Start sending triggers on the next win flip and continue until stopped by stopTrigger Triggers start 1 frame after the frame on which the first trigger is sent.

Example:

```
bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111)
bits.startTrigger()
while imageOn:
    #do some processing
    continue
bits.stopTrigger()
bits.win.flip()
```

statusBoxAddKeys (*map*)

Add key mappings to an existing map. statusBox events can be mapped to a number of physical events on Bits# They can be mapped to digital input lines, triggers and CB6 IR input channels. The format for map is a list of tuples with each tuple containing the name of the RTBox button to be mapped and its source eg ('btn1','Din1') maps physical input Din1 to logical button btn1. statusBox has 23 logical buttons (btn1-23). Unlike RTBox buttons/events can only be partially mapped to multiple physical inputs. That is a logical button can be mapped to more than 1 physical input but a physical input can only be mapped to 1 logical button. So, this function over write any existing mappings if the physical input is the same.

Example:

```
bits.RTBoxSetKeys([('btn1', 'Din0'), ('btn2', 'Din1')])
bits.RTBoxAddKeys([('btn1', 'IRButtonA'), ('btn2', 'IRButtonB')])
```

Will link Din0 to button 1 and Din1 to button 2. Then adds IRButtonA and IRButtonB alongside the original mappings.

Now both hard wired and IR inputs will emulate the same logical button press.

To match with the CRS hardware description inputs are labelled as follows.

TrigIn, Din0 ... Din9, IRButtonA ... IRButtonF, AnalogIn1 ... AnalogIn6

Logical buttons are numbered from 1 to 23.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

statusBoxDisable ()

Disables the detection of statusBox events. This is useful to stop the Bits# from reporting key presses When you no longer need them. And must be done before using any other data logging methods.

It undoes any button - input mappings

statusBoxEnable (mode=None, map=None, threshold=None)

Sets up the stautsBox with preset or bespoke mappings and enables event detection.

stautsBox events can be mapped to a number of physical events on Bits# They can be mapped to digital input lines, tigers and CB6 IR input channels.

mode is a list of strings. Preset mappings provided via mode:

- CB6 for the CRS CB6 IR response box connected mapped to btn1-6
- IO for a three button box connected to Din0-2 mapped to btn1-3
- IO6 for a six button box connected to Din0-5 mapped to btn1-6
- IO10 for a ten button box connected to Din0-9 mapped to btn1-10
- Trigger maps the trigIn to btn17
- Analog maps the 6 analog inputs on a Bits# to btn18-23

If CB6 and IOx are used together the Dins are mapped from btn7 onwards.

If mode = None or is not set then the value of self.statusBoxMode is used.

Bespoke Mappings overwrite preset ones.

The format for map is a list of tuples with each tuple containing the name of the button to be mapped and its source eg ('btn1','Din0') maps physical input Din0 to logical button btn1.

Note the lowest number button event is Btn1

statusBox has 23 logical buttons (btn1-123). Buttons/events can be mapped to multiple physical inputs and stay mapped until reset.

mode is a string or list of strings that contains keywords to determine present mappings and modes for statusBox.

If mode includes 'Down' button events will be detected when pressed. If mode includes 'Up' button events will be detected when released. You can detect both types of event noting that the event detector will look for transitions and ignore what it sees as the starting state.

To match with the CRS hardware description inputs are labelled as follows.

TrigIn, Din0 ... Din9, IRButtonA ... IRButtonF, AnalogIn1 ... AnalogIn6

Logical buttons are numbered from 1 to 23.

threshold sets the threshold by which analog inputs must change to trigger a button press event. If None the threshold will be set very high so that no such events are triggered. Analog inputs must cycle up and down by threshold to be detected as separate events. So if only 'Up' events are detected the input must go up by threshold, then come down again and then go back up to register 2 up events.

Example:

```
bits.statusBoxEnable(mode = 'Down'), map = [('btn1', 'Din0'), ('btn2', 'Din1')]
```

enables the statusBox to detect Down events on buttons 1 and 2 where they are mapped to DIN0 and DIN1.

Example:

```
bits.statusBoxEnable(mode = ['Down', 'CB6'])
```

enables the status Box emulation to detect Down events on the standard CB6 IR response box keys.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

statusBoxKeysPressed (*N=1*)

Check to see if (at least) the appropriate number of RTBox style key presses have been made.

Example

```
bits.statusBoxKeysPressed(5)
```

will return false until 5 button presses have been recorded.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

statusBoxResetKeys ()

statusBoxSetKeys (*map*)

Set key mappings: first resets existing then adds new ones. Does not reset any event that is not in the new list. statusBox events can be mapped to a number of physical events on Bits# They can be mapped to digital input lines, triggers and CB6 IR input channels. The format for map is a list of tuples with each tuple containing the name of the RTBox button to be mapped and its source eg ('btn1', 'Din1') maps physical input Din1 to logical button btn1.

statusBox has 17 logical buttons (btn1-17) Buttons/events can be mapped to multiple physical inputs and stay mapped until reset.

Example

```
bits.RTBoxSetKeys([('btn1','Din0'),('btn2','IRButtonA']))
```

Will link physical Din0 to logical button 1 and IRButtonA to button 2.

To match with the CRS hardware description inputs are labelled as follows.

TrigIn, Din0 ... Din9, IRButtonA ... IRButtonF, AnalogIn1 ... AnalogIn6

Logical buttons are numbered from 1 to 23.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

statusBoxWait ()

Waits until (at least) one of RTBox style key presses have been made Pauses program execution in mean time.

Example

```
res = bits.statusBoxWait()
```

will suspend all other activity until 1 button press has been recorded and will then return a dict / structure containing results.

Results can be accessed as follows:

structure res.dir, res.button, res.time

or dictionary res['dir'], res['button'], res['time']

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also DBits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

statusBoxWaitN (N=1)

Waits until (at least) the appropriate number of RTBox style key presses have been made Pauses program execution in mean time.

Example

```
res = bits.statusBoxWaitN(5)
```

will suspend all other activity until 5 button presses have been recorded and will then return a list of Dicts containing the 5 results.

Results can be accessed as follows:

structure:

```
res[0].dir, res[0].button, res[0].time
```

or dictionary:

```
res[0]['dir'], res[0]['button'], res[0]['time']
```

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

stop()

[Not currently implemented] Used to stop event collection by the device.

Not really needed as other members now do this.

stopAnalog()

will stop sending analogs signals at the next win flip.

Example:

```
bits.set Analog(4.5, -2.2)
bits.startAnalog()
bits.win.flip()
while not response:
    #do some processing.
    bits.win.flip()
bits.stopAnalog()
bits.win.flip()
```

stopGoggles()

Stop the stereo goggles from toggling

Example:

```
bits.startGoggles(0, 1)
bits.win.flip()
while not response
    bits.win.flip()
    #do some processing
bits.stopGoggles()
bits.win.flip()
```

Starts toggling the goggles with the right eye open in sync with the first win.flip(0) within the loop. The open eye will alternate.

Note it is safer to leave the goggles toggling forever, ie to never call stopGoggles().

stopStatusLog()

Stop logging data from the Bits# and extracts the raw status values and significant events and puts them in statusValues and statusEvents.

statusValues will end up containing dict like objects of the following style: *sample, time, trigIn, DIN[10], DWORD, IR[6], ADC[6]*

They can be accessed as statusValues[i]['sample'] or statusValues[i].sample, statusValues[x].ADC[j].

StatusEvents will end up containing dict like objects of the following style: *source, input, direction, time*

The data can be accessed as statusEvents[i]['time'] or statusEvents[i].time

Waits for _statusLog to finish properly so can introduce a timing delay.

Example:

```
bits.startStatusLog()
while not event
    #do some processing
    continue
bits.stopStatusLog()
print(bits.statusValues[0].time)
print(bits.statusEvents[0].time)
```

Will display the time stamps of the first status value recorded and the first meaningful event.

Note that the firmware in Bits# units varies over time and some features of this class may not work for all firmware versions. Also Bits# units can be configured in various ways via their config.xml file so this class makes certain assumptions about the configuration. In particular it is assumed that all digital inputs, triggers and analog inputs are reported as part of status updates. If some of these report are disabled in your config.xml file then 'status' and 'event' commands in this class may not work.

stopTrigger()

Stop sending triggers at the next win flip.

Example:

```
bits.setTrigger(0b0000000010, 2.0, 4.0, 0b0111111111)
bits.startTrigger()
while imageOn:
    #do some processing
    continue
bits.stopTrigger()
bits.win.flip()
```

syncClocks(t)

Synchronise the Bits/RTBox Clock with the host clock Given by t.

property temporalDithering

Temporal dithering can be set to True or False

property win

The window that this box is attached to

For display calibration

ColorCAL

Attributes

ColorCAL([port, maxAttempts])

A class to handle the CRS Ltd ColorCAL device

Details

class psychopy.hardware.crs.colorcal.ColorCAL (*port=None, maxAttempts=2*)

A class to handle the CRS Ltd ColorCAL device

Open serial port connection with Colorcal II device

Usage cc = ColorCAL(port, maxAttempts)

If no port is provided then the following defaults will be tried:

- /dev/cu.usbmodem0001 (OSX)
- /dev/ttyACM0
- COM3 (windows)

calibrateZero ()

Perform a calibration to zero light.

For early versions of the ColorCAL this had to be called after connecting to the device. For later versions the dark calibration was performed at the factory and stored in non-volatile memory.

You can check if you need to run a calibration with:

```
ColorCAL.getNeedsCalibrateZero()
```

driverFor = ['colorcal']

getCalibMatrix ()

Get the calibration matrix from the device, needed for transforming measurements into real-world values.

This is normally retrieved during `__init__` and stored as `ColorCal.calibMatrix` so most users don't need to call this function.

getInfo ()

Queries the device for information

usage::

```
(ok, serialNumber, firmwareVersion, firmwareBuild) = colorCal.getInfo()
```

ok will be True/False Other values will be a string or None.

getLum ()

Conducts a measurement and returns the measured luminance

Note: The luminance is always also stored as `.lastLum`

getNeedsCalibrateZero ()

Check whether the device needs a dark calibration

In initial versions of CRS ColorCAL mkII the device stored its zero calibration in volatile memory and needed to be calibrated in darkness each time you connected it to the USB

This function will check whether your device requires that (based on firmware build number and whether you've already done it since python connected to the device).

Returns True or False

longName = 'CRS ColorCAL'

measure ()

Conduct a measurement and return the X,Y,Z values

Usage:

```
ok, X, Y, Z = colorCal.measure()
```

Where: ok is True/False X, Y, Z are the CIE coordinates (Y is luminance in cd/m**2)

Following a call to measure, the values ColorCAL.lastLum will also be populated with, for compatibility with other devices used by PsychoPy (notably the PR650/PR655)

readline (*size=None, eol='\n\r'*)

This should be used in place of the standard serial.Serial.readline() because that doesn't allow us to set the eol character

sendMessage (*message, timeout=0.1*)

Send a command to the photometer and wait an allotted timeout for a response.

9.5.8 egi (pynetstation)

Support for egi is now provided via the [egi-pynetstation](#) third-party library. This is included within Standalone PsychoPy from 2022.2.0

See the [`egi-pynetstation documentation<https://egi-pynetstation.readthedocs.io/en/latest/>`](https://egi-pynetstation.readthedocs.io/en/latest/) for further details.

9.5.9 Launch an fMRI experiment: Test or Scan

Idea: Run or debug an experiment script using exactly the same code, i.e., for both testing and online data acquisition. To debug timing, you can emulate sync pulses and user responses. Limitations: pyglet only; keyboard events only.

class psychopy.hardware.emulator.**ResponseEmulator** (*simResponses=None*)

Class to allow simulation of a user's keyboard responses during a scan.

Given a list of response tuples (time, key), the thread will simulate a user pressing a key at a specific time (relative to the start of the run).

Author: Jeremy Gray; Idea: Mike MacAskill

_delete ()

Remove current thread from the dict of currently running threads.

_set_tstate_lock ()

Set a lock object which will be released by the interpreter when the underlying thread state (see pystate.h) gets deleted.

property daemon

A boolean value indicating whether this thread is a daemon thread.

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when only daemon threads are left.

property ident

Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the `get_ident()` function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isAlive()

Return whether the thread is alive.

This method is deprecated, use `is_alive()` instead.

is_alive()

Return whether the thread is alive.

This method returns `True` just before the `run()` method starts until just after the `run()` method terminates. The module function `enumerate()` returns a list of all alive threads.

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose `join()` method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As `join()` always returns `None`, you must call `is_alive()` after `join()` to decide whether a timeout happened – if the thread is still alive, the `join()` call timed out.

When the timeout argument is not present or `None`, the operation will block until the thread terminates.

A thread can be `join()`ed many times.

`join()` raises a `RuntimeError` if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to `join()` a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or `None` if it has not been started.

This is a non-negative integer. See the `get_native_id()` function. This represents the Thread ID as reported by the kernel.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's `run()` method to be invoked in a separate thread of control.

This method will raise a `RuntimeError` if called more than once on the same thread object.

class `psychopy.hardware.emulator.SyncGenerator` (*TR=1.0, TA=1.0, volumes=10, sync='5', skip=0, sound=False, **kwargs*)

Class for a character-emitting metronome thread (emulate MR sync pulse).

Aim: Allow testing of temporal robustness of fMRI scripts by emulating a hardware sync pulse. Adds an arbitrary ‘sync’ character to the key buffer, with sub-millisecond precision (less precise if CPU is maxed). Recommend: TR=1.000 or higher and less than 100% CPU. Shorter TR -> higher CPU load.

Parameters

- **TR** – seconds between volume acquisitions
- **TA** – seconds to acquire one volume
- **volumes** – number of 3D volumes to obtain in a given scanning run
- **sync** – character used as flag for sync timing, default='5'
- **skip** – how many frames to silently omit initially during T1 stabilization, no sync pulse. Not needed to test script timing, but will give more accurate feel to start of run. aka “disc-dacqs”.
- **sound** – simulate scanner noise

`_delete()`

Remove current thread from the dict of currently running threads.

`_set_tstate_lock()`

Set a lock object which will be released by the interpreter when the underlying thread state (see `pystate.h`) gets deleted.

`property daemon`

A boolean value indicating whether this thread is a daemon thread.

This must be set before `start()` is called, otherwise `RuntimeError` is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to `daemon = False`.

The entire Python program exits when only daemon threads are left.

`property ident`

Thread identifier of this thread or `None` if it has not been started.

This is a nonzero integer. See the `get_ident()` function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

`isAlive()`

Return whether the thread is alive.

This method is deprecated, use `is_alive()` instead.

`is_alive()`

Return whether the thread is alive.

This method returns `True` just before the `run()` method starts until just after the `run()` method terminates. The module function `enumerate()` returns a list of all alive threads.

`join(timeout=None)`

Wait until the thread terminates.

This blocks the calling thread until the thread whose `join()` method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not `None`, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As `join()` always returns `None`, you must call `is_alive()` after `join()` to decide whether a timeout happened – if the thread is still alive, the `join()` call timed out.

When the timeout argument is not present or `None`, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

property name

A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

property native_id

Native integral thread ID of this thread, or None if it has not been started.

This is a non-negative integer. See the get_native_id() function. This represents the Thread ID as reported by the kernel.

run ()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

start ()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

```
psychopy.hardware.emulator.launchScan(win, settings, globalClock=None, simRe-  
sponses=None, mode=None, esc_key='escape',  
instr='select Scan or Test, press enter',  
wait_msg='waiting for scanner...', wait_timeout=300,  
log=True)
```

Accepts up to four fMRI scan parameters (TR, volumes, sync-key, skip), and launches an experiment in one of two modes: Scan, or Test.

Usage See Coder Demo -> experiment control -> fMRI_launchScan.py.

In brief: 1) from psychopy.hardware.emulator import launchScan; 2) Define your args; and 3) add 'vol = launchScan(args)' at the top of your experiment script.

launchScan() waits for the first sync pulse and then returns, allowing your experiment script to proceed. The key feature is that, in test mode, it first starts an autonomous thread that emulates sync pulses (i.e., emulated by your CPU rather than generated by an MRI machine). The thread places a character in the key buffer, exactly like a keyboard event does. launchScan will wait for the first such sync pulse (i.e., character in the key buffer). launchScan returns the number of sync pulses detected so far (i.e., 1), so that a script can account for them explicitly.

If a globalClock is given (highly recommended), it is reset to 0.0 when the first sync pulse is detected. If a mode was not specified when calling launchScan, the operator is prompted to select Scan or Test.

If **scan mode** is selected, the script will wait until the first scan pulse is detected. Typically this would be coming from the scanner, but note that it could also be a person manually pressing that key.

If **test mode** is selected, launchScan() starts a separate thread to emit sync pulses / key presses. Note that this thread is effectively nothing more than a key-pressing metronome, emitting a key at the start of every TR, doing so with high temporal precision.

If your MR hardware interface does not deliver a key character as a sync flag, you can still use `launchScan()` to test script timing. You have to code your experiment to trigger on either a sync character (to test timing) or your usual sync flag (for actual scanning).

Parameters win: a *Window* object (required)

settings [a dict containing up to 5 parameters] (2 required: TR, volumes)

TR : seconds per whole-brain volume (minimum value = 0.1s)

volumes : number of whole-brain (3D) volumes to obtain in a given scanning run.

sync : (optional) key for sync timing, default = '5'.

skip : (optional) how many volumes to silently omit initially (during T1 stabilization, no sync pulse). default = 0.

sound : (optional) whether to play a sound when simulating scanner sync pulses

globalClock : optional but highly recommended *Clock* to be used during the scan; if one is given, it is reset to 0.000 when the first sync pulse is received.

simResponses : optional list of tuples [(time, key), (time, key), ...]. time values are seconds after the first scan pulse is received.

esc_key : key to be used for user-interrupt during launch. default = 'escape'

mode : if mode is 'Test' or 'Scan', `launchScan()` will start in that mode.

instr : instructions to be displayed to the scan operator during mode selection.

wait_msg : message to be displayed to the subject while waiting for the scan to start (i.e., after operator indicates start but before the first scan pulse is received).

wait_timeout : time in seconds that `launchScan` will wait before assuming something went wrong and exiting. Defaults to 300sec (5 min). Raises a `RuntimeError` if no sync pulse is received in the allowable time.

class `psychoPy.hardware.emulator.ResponseEmulator` (*simResponses=None*)

Class to allow simulation of a user's keyboard responses during a scan.

Given a list of response tuples (time, key), the thread will simulate a user pressing a key at a specific time (relative to the start of the run).

Author: Jeremy Gray; Idea: Mike MacAskill

run ()

Method representing the thread's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

class `psychoPy.hardware.emulator.SyncGenerator` (*TR=1.0, TA=1.0, volumes=10, sync='5', skip=0, sound=False, **kwargs*)

Class for a character-emitting metronome thread (emulate MR sync pulse).

Aim: Allow testing of temporal robustness of fMRI scripts by emulating a hardware sync pulse. Adds an arbitrary 'sync' character to the key buffer, with sub-millisecond precision (less precise if CPU is maxed). Recommend: TR=1.000 or higher and less than 100% CPU. Shorter TR -> higher CPU load.

Parameters

- **TR** – seconds between volume acquisitions
- **TA** – seconds to acquire one volume

- **volumes** – number of 3D volumes to obtain in a given scanning run
- **sync** – character used as flag for sync timing, default='5'
- **skip** – how many frames to silently omit initially during T1 stabilization, no sync pulse. Not needed to test script timing, but will give more accurate feel to start of run. aka “disc-dacqs”.
- **sound** – simulate scanner noise

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

9.5.10 fORP response box

fORP fibre optic (MR-compatible) response devices by CurrentDesigns: <http://www.curdes.com/> This class is only useful when the fORP is connected via the serial port.

If you're connecting via USB, just treat it like a standard keyboard. E.g., use a Keyboard component, and typically listen for Allowed keys '1', '2', '3', '4', '5'. Or use `event.getKeys()`.

class `psychopy.hardware.forp.ButtonBox` (*serialPort=1, baudrate=19200*)

Serial line interface to the fORP MRI response box.

To use this object class, select the box use setting *serialPort*, and connect the serial line. To emulate key presses with a serial connection, use `getEvents(asKeys=True)` (e.g., to be able to use a RatingScale object during scanning). Alternatively connect the USB cable and use fORP to emulate a keyboard.

fORP sends characters at 800Hz, so you should check the buffer frequently. Also note that the trigger event numpy the fORP is typically extremely short (occurs for a single 800Hz epoch).

Parameters

serialPort : should be a number (where 1=COM1, ...)

baud : the communication rate (baud), eg, 57600

classmethod `_decodePress` (*pressCode*)

Returns a list of buttons and whether they're pressed, given a character code.

pressCode : A number with a bit set for every button currently pressed. Will be between 0 and 31.

_generateEvents (*pressCode*)

For a given button press, returns a list buttons that went from unpressed to pressed. Also flags any unpressed buttons as unpressed.

pressCode : a number with a bit set for every button currently pressed.

clearBuffer ()

Empty the input buffer of all characters

clearStatus ()

Resets the pressed statuses, so `getEvents` will return pressed buttons, even if they were already pressed in the last call.

getEvents () (*stored as ForpBox.rawEvts*)

returnRaw : return (not just store) the full event list

asKeys : If True, will also emulate pygame keyboard events, so that button 1 will register as a keyboard event with value “1”, and as such will be detectable using `event.getKeys()`

allowRepeats : If True, this will return pressed buttons even if they were held down between calls to `getEvents()`. If the fORP is on the “Eprime” setting, you will get a stream of button presses while a button is held down. On the “Bitwise” setting, you will get a set of all currently pressed buttons every time a button is pressed or released. This option might be useful if you think your participant may be holding the button down before you start checking for presses.

getUniqueEvents (*fullEvts=False*)

Returns a Python set of the unique (unordered) events of either a list given or the current rawEvts buffer

9.5.11 iolab

The ioLab button box has not been made for many years now (and the company no longer exists) so we have withdrawn support for this device in . We recommend you get a more modern device, such as the [LabHackers Millikey](#)

9.5.12 joystick (pygame and pygame)

AT THE MOMENT JOYSTICK DOES NOT APPEAR TO WORK UNDER PYGLET. We need someone motivated and capable to go and get this right (problem is with event polling under pygame)

Control joysticks and gamepads from within PsychoPy.

You do need a window (and you need to be flipping it) for the joystick to be updated.

Known issues:

- currently under pygame the joystick axes initialise to a value of zero and stay like this until the first time that axis moves
- currently pygame (1.9.1) spits out lots of debug messages about the joystick and these can’t be turned off :-/

Typical usage:

```
from psychopy.hardware import joystick
from psychopy import visual

joystick.backend='pygame' # must match the Window
win = visual.Window([400,400], winType='pygame')

nJoys = joystick.getNumJoysticks() # to check if we have any
id = 0
joy = joystick.Joystick(id) # id must be <= nJoys - 1

nAxes = joy.getNumAxes() # for interest
while True: # while presenting stimuli
    joy.getX()
    # ...
    win.flip() # flipping implicitly updates the joystick info
```

class psychopy.hardware.joystick.XboxController (*id, *args, **kwargs*)

Joystick template class for the XBox 360 controller.

Usage:

```
xbctrl = XboxController(0) # joystick ID
y_btn_state = xbctrl.y # get the state of the ‘Y’ button
```

An object to control a multi-axis joystick or gamepad.

Known issues Currently under pygame backends the axis values initialise to zero rather than reading the current true value. This gets fixed on the first change to each axis.

`_clip_range` (*val*)

Clip the range of a value between -1.0 and +1.0. Needed for joystick axes.

Parameters *val* –

Returns

`get_a` ()

Get the 'A' button state.

Returns bool, True if pressed down

`get_b` ()

Get the 'B' button state.

Returns bool, True if pressed down

`get_back` ()

Get 'back' button state (button to the right of the left joystick).

Returns bool, True if pressed down

`get_hat_axis` ()

Get the states of the hat (sometimes called the 'directional pad'). The hat can only indicate direction but not displacement.

This function reports hat values in the same way as a joystick so it may be used interchangeably with existing analog joystick code.

Returns a tuple (X,Y) indicating which direction the hat is pressed between -1.0 and +1.0. Positive values indicate presses in the right or up direction.

Returns tuple, zero centered X, Y values.

`get_left_shoulder` ()

Get left 'shoulder' trigger state.

Returns bool, True if pressed down

`get_left_thumbstick` ()

Get the state of the left joystick button; activated by pressing down on the stick.

Returns bool, True if pressed down

`get_left_thumbstick_axis` ()

Get the axis displacement values of the left thumbstick.

Returns a tuple (X,Y) indicating thumbstick displacement between -1.0 and +1.0. Positive values indicate the stick is displaced right or up.

Returns tuple, zero centered X, Y values.

`get_named_buttons` (*button_names*)

Get the states of multiple buttons using names. A list of button states is returned for each string in list 'names'.

Parameters *button_names* – tuple or list of button names

Returns

get_right_shoulder ()

Get right 'shoulder' trigger state.

Returns bool, True if pressed down

get_right_thumbstick ()

Get the state of the right joystick button; activated by pressing down on the stick.

Returns bool, True if pressed down

get_right_thumbstick_axis ()

Get the axis displacement values of the right thumbstick.

Returns a tuple (X,Y) indicating thumbstick displacement between -1.0 and +1.0. Positive values indicate the stick is displaced right or up.

Returns tuple, zero centered X, Y values.

get_start ()

Get 'start' button state (button to the left of the 'X' button).

Returns bool, True if pressed down

get_trigger_axis ()

Get the axis displacement values of both index triggers.

Returns a tuple (L,R) indicating index trigger displacement between -1.0 and +1.0. Values increase from -1.0 to 1.0 the further a trigger is pushed.

Returns tuple, zero centered L, R values.

get_x ()

Get the 'X' button state.

Returns bool, True if pressed down

get_y ()

Get the 'Y' button state.

Returns bool, True if pressed down

`psychopy.hardware.joystick.getNumJoysticks ()`

Return a count of the number of joysticks available.

class `psychopy.hardware.joystick.Joystick (id)`

An object to control a multi-axis joystick or gamepad.

Known issues Currently under pyglet backends the axis values initialise to zero rather than reading the current true value. This gets fixed on the first change to each axis.

getAllAxes ()

Get a list of all current axis values.

getAllButtons ()

Get the state of all buttons as a list.

getAllHats ()

Get the current values of all available hats as a list of tuples.

Each value is a tuple (x, y) where x and y can be -1, 0, +1

getAxis (axisId)

Get the value of an axis by an integer id.

(from 0 to number of axes - 1)

getButton (*buttonId*)

Get the state of a given button.

buttonId should be a value from 0 to the number of buttons-1

getHat (*hatId=0*)

Get the position of a particular hat.

The position returned is an (x, y) tuple where x and y can be -1, 0 or +1

getName ()

Return the manufacturer-defined name describing the device.

getNumAxes ()

Return the number of joystick axes found.

getNumButtons ()

Return the number of digital buttons on the device.

getNumHats ()

Get the number of hats on this joystick.

The GLFW backend makes no distinction between hats and buttons. Calling 'getNumHats()' will return 0.

getX ()

Return the X axis value (equivalent to joystick.getAxis(0)).

getY ()

Return the Y axis value (equivalent to joystick.getAxis(1)).

getZ ()

Return the Z axis value (equivalent to joystick.getAxis(2)).

9.5.13 labjacks (USB I/O devices)

provides an interface to the labjack U3 class with a couple of minor additions.

This is accessible by:

```
from psychopy.hardware.labjacks import U3
```

Except for the additional *setdata* function the U3 class operates exactly as that in the U3 library that labjack provides, documented here:

<http://labjack.com/support/labjackpython>

Note: To use labjack devices you do need also to install the driver software described on the page above

9.5.14 Minolta

Minolta light-measuring devices See <http://www.konicaminolta.com/instruments>

class psychopy.hardware.minolta.CS100A (port, maxAttempts=1)

A class to define a Minolta CS100A photometer

You need to connect a CS100A to the serial (RS232) port and **when you turn it on press the F key** on the device. This will put it into the correct mode to communicate with the serial port.

usage:

```
from psychopy.hardware import minolta
phot = minolta.CS100A(port)
if phot.OK: # then we successfully made a connection
    print(phot.getLum())
```

Parameters port: string

the serial port that should be checked

maxAttempts: int If the device doesn't respond first time how many attempts should be made?

If you're certain that this is the correct port and the device is on and correctly configured then this could be set high. If not then set this low.

Troubleshooting Various messages are printed to the log regarding the function of this device, but to see them you need to set the printing of the log to the correct level:

```
from psychopy import logging
logging.console.setLevel(logging.ERROR) # error messages only
logging.console.setLevel(logging.INFO) # more info
logging.console.setLevel(logging.DEBUG) # log all communications
```

If you're using a keyspan adapter (at least on macOS) be aware that it needs a driver installed. Otherwise no ports will be found.

Error messages:

ERROR: Couldn't connect to Minolta CS100A on ____: This likely means that the device is not connected to that port (although the port has been found and opened). Check that the device has the / in the bottom right of the display; if not turn off and on again holding the F key.

ERROR: No reply from CS100A: The port was found, the connection was made and an initial command worked, but then the device stopped communicating. If the first measurement taken with the device after connecting does not yield a reasonable intensity the device can sulk (not a technical term!). The "[" on the display will disappear and you can no longer communicate with the device. Turn it off and on again (with F depressed) and use a reasonably bright screen for your first measurement. Subsequent measurements can be dark (or we really would be in trouble!).

checkOK (msg)

Check that the message from the photometer is OK. If there's an error show it (printed).

Then return True (OK) or False.

clearMemory ()

Clear the memory of the device from previous measurements

getLum()

Makes a measurement and returns the luminance value

measure()

Measure the current luminance and set `.lastLum` to this value

sendMessage (*message, timeout=5.0*)

Send a command to the photometer and wait an allotted timeout for a response.

The message can be in either bytes or unicode but the returned string will always be utf-encoded.

setMaxAttempts (*maxAttempts*)

Changes the number of attempts to send a message and read the output. Typically this should be low initially, if you aren't sure that the device is setup correctly but then, after the first successful reading, set it higher.

setMode (*mode='04'*)

Set the mode for measurements. Returns True (success) or False

'04' means absolute measurements. '08' = peak '09' = cont

See user manual for other modes

class `psychopy.hardware.minolta.LS100` (*port, maxAttempts=1*)

A class to define a Minolta LS100 (or LS110?) photometer

You need to connect a LS100 to the serial (RS232) port and **when you turn it on press the F key** on the device. This will put it into the correct mode to communicate with the serial port.

usage:

```
from psychopy.hardware import minolta
phot = minolta.LS100(port)
if phot.OK: # then we successfully made a connection
    print(phot.getLum())
```

Parameters `port`: string

the serial port that should be checked

maxAttempts: int If the device doesn't respond first time how many attempts should be made?

If you're certain that this is the correct port and the device is on and correctly configured then this could be set high. If not then set this low.

Troubleshooting Various messages are printed to the log regarding the function of this device, but to see them you need to set the printing of the log to the correct level:

```
from psychopy import logging
logging.console.setLevel(logging.ERROR) # error messages only
logging.console.setLevel(logging.INFO) # more info
logging.console.setLevel(logging.DEBUG) # log all communications
```

If you're using a keyspan adapter (at least on macOS) be aware that it needs a driver installed. Otherwise no ports will be found.

Error messages:

ERROR: Couldn't connect to Minolta LS100/110 on ____: This likely means that the device is not connected to that port (although the port has been found and opened). Check that the device has the / in the bottom right of the display; if not turn off and on again holding the *F* key.

ERROR: No reply from LS100: The port was found, the connection was made and an initial command worked, but then the device stopped communicating. If the first measurement taken with the device after connecting does not yield a reasonable intensity the device can sulk (not a technical term!). The “[” on the display will disappear and you can no longer communicate with the device. Turn it off and on again (with F depressed) and use a reasonably bright screen for your first measurement. Subsequent measurements can be dark (or we really would be in trouble!!).

checkOK (*msg*)

Check that the message from the photometer is OK. If there’s an error show it (printed).

Then return True (OK) or False.

clearMemory ()

Clear the memory of the device from previous measurements

getLum ()

Makes a measurement and returns the luminance value

measure ()

Measure the current luminance and set .lastLum to this value

sendMessage (*message, timeout=5.0*)

Send a command to the photometer and wait an allotted timeout for a response.

The message can be in either bytes or unicode but the returned string will always be utf-encoded.

setMaxAttempts (*maxAttempts*)

Changes the number of attempts to send a message and read the output. Typically this should be low initially, if you aren’t sure that the device is setup correctly but then, after the first successful reading, set it higher.

setMode (*mode='04'*)

Set the mode for measurements. Returns True (success) or False

‘04’ means absolute measurements. ‘08’ = peak ‘09’ = cont

See user manual for other modes

9.5.15 PhotoResearch

Supported devices:

- *PR650*
- *PR655/PR670*

PhotoResearch spectrophotometers See <http://www.photoresearch.com/>

class psychopy.hardware.pr.PR650 (*port, verbose=None*)

An interface to the PR650 via the serial port.

(Added in version 1.63.02)

example usage:

```

from psychopy.hardware.pr import PR650
myPR650 = PR650(port)
myPR650.getLum() # make a measurement
nm, power = myPR650.getLastSpectrum() # get a power spectrum for the
last measurement
    
```

NB `psychopy.hardware.findPhotometer()` will locate and return any supported device for you so you can also do:

```

from psychopy import hardware
phot = hardware.findPhotometer()
print(phot.getLum())
    
```

Troubleshooting Various messages are printed to the log regarding the function of this device, but to see them you need to set the printing of the log to the correct level:

```

from psychopy import logging
logging.console.setLevel(logging.ERROR) # error messages only
logging.console.setLevel(logging.INFO) # will give more info
logging.console.setLevel(logging.DEBUG) # log all communications
    
```

If you're using a keyspan adapter (at least on macOS) be aware that it needs a driver installed. Otherwise no ports will be found.

Also note that the attempt to connect to the PR650 must occur within the first few seconds after turning it on.

getLastLum()

This retrieves the luminance (in cd/m^2) from the last call to `.measure()`

getLastSpectrum(parse=True)

This retrieves the spectrum from the last call to `.measure()`

If `parse=True` (default): The format is a num array with 100 rows [nm, power]

otherwise: The output will be the raw string from the PR650 and should then be passed to `.parseSpectrumOutput()`. It's more efficient to parse R,G,B strings at once than each individually.

getLum()

Makes a measurement and returns the luminance value

getSpectrum(parse=True)

Makes a measurement and returns the current power spectrum

If parse=True (default): The format is a num array with 100 rows [nm, power]

If parse=False (default): The output will be the raw string from the PR650 and should then be passed to `.parseSpectrumOutput()`. It's slightly more efficient to parse R,G,B strings at once than each individually.

measure(timeOut=30.0)

Make a measurement with the device. For a PR650 the device is instructed to make a measurement and then subsequent commands are issued to retrieve info about that measurement.

parseSpectrumOutput(rawStr)

Parses the strings from the PR650 as received after sending the command 'd5'. The input argument "rawStr" can be the output from a single phosphor spectrum measurement or a list of 3 such measurements [rawR, rawG, rawB].

sendMessage (*message, timeout=0.5, DEBUG=False*)

Send a command to the photometer and wait an allotted timeout for a response (Timeout should be long for low light measurements)

class psychopy.hardware.pr.PR655 (*port*)

An interface to the PR655/PR670 via the serial port.

example usage:

```
from psychopy.hardware.pr import PR655
myPR655 = PR655(port)
myPR655.getLum() # make a measurement
nm, power = myPR655.getLastSpectrum() # get a power spectrum for the
last measurement
```

NB `psychopy.hardware.findPhotometer()` will locate and return any supported device for you so you can also do:

```
from psychopy import hardware
phot = hardware.findPhotometer()
print(phot.getLum())
```

Troubleshooting If the device isn't responding try turning it off and turning it on again, and/or disconnecting/reconnecting the USB cable. It may be that the port has become controlled by some other program.

endRemoteMode ()

Puts the colorimeter back into normal mode

getDeviceSN ()

Return the device serial number

getDeviceType ()

Return the device type (e.g. 'PR-655' or 'PR-670')

getLastColorTemp ()

Fetches (from the device) the color temperature (K) of the last measurement

Returns list: status, units, exponent, correlated color temp (Kelvins), CIE 1960 deviation

See also `measure()` automatically populates `pr655.lastColorTemp` with the color temp in Kelvins

getLastSpectrum (*parse=True*)

This retrieves the spectrum from the last call to `measure()`

If *parse=True* (default):

The format is a num array with 100 rows [nm, power]

otherwise:

The output will be the raw string from the PR650 and should then be passed to `parseSpectrumOutput()`. It's more efficient to parse R,G,B strings at once than each individually.

getLastTristim ()

Fetches (from the device) the last CIE 1931 Tristimulus values

Returns list: status, units, Tristimulus Values

See also `measure()` automatically populates `pr655.lastTristim` with just the tristimulus coordinates

getLastUV()

Fetches (from the device) the last CIE 1976 u,v coords

Returns list: status, units, Photometric brightness, u, v

See also `measure()` automatically populates `pr655.lastUV` with [u,v]

getLastXY()

Fetches (from the device) the last CIE 1931 x,y coords

Returns list: status, units, Photometric brightness, x,y

See also `measure()` automatically populates `pr655.lastXY` with [x,y]

measure(timeOut=30.0)

Make a measurement with the device.

This automatically populates:

- `.lastLum`
- `.lastSpectrum`
- `.lastCIExy`
- `.lastCIEuv`

parseSpectrumOutput(rawStr)

Parses the strings from the PR650 as received after sending the command 'D5'. The input argument "rawStr" can be the output from a single phosphor spectrum measurement or a list of 3 such measurements [rawR, rawG, rawB].

startRemoteMode()

Sets the Colorimeter into remote mode

9.5.16 pylink (SR Research)

For now the SR Research `pylink` module is packaged with the Standalone flavours of and can be imported with:

```
import pylink
```

You do need to install the Display Software (which they also call Eyelink Developers Kit) for your particular platform. This can be found by following the threads from the SR Research support forum (creating an account is required):

<https://www.sr-support.com/thread-13.html>

For documentation of `pylink`, see:

<https://www.sr-support.com/thread-48.html>

9.5.17 pump - A simple interface to the Cetoni neMESYS syringe pump system

Please specify the name of the pump configuration to use in the preferences under Hardware / Qmix pump configuration. See the [readme file](#) of the `pyqmix` project for details on how to set up your computer and create the configuration file.

Simple interface to the Cetoni neMESYS syringe pump system, based on the `pyqmix` library. The syringe pump system is described in the following publication:

CA Andersen, L Alfine, K Ohla, & R Höchenberger (2018): “A new gustometer: Template for the construction of a portable and modular stimulator for taste and lingual touch.” *Behavior Research Methods*. doi: 10.3758/s13428-018-1145-1

```
class psychopy.hardware.qmix.Pump(index, volumeUnit='mL', flowRateUnit='mL/s', syringeType='50 mL glass')
```

An interface to Cetoni neMESYS syringe pumps, based on the `pyqmix` library.

Parameters

- **index** (*int*) – The index of the pump. The first pump in the system has *index=0*, the second *index=1*, etc.
- **volumeUnit** ('mL') – The unit in which the volumes are provided. Currently, only 'ml' is supported.
- **flowRateUnit** ('mL/s' or 'mL/min') – The unit in which flow rates are provided.
- **syringeType** ('25 mL glass' or '50 mL glass') – Type of the installed syringe, as understood by `pyqmix`.

```
aspirate(volume, flowRate, waitUntilDone=False, switchValveWhenDone=False)
```

Aspirate the specified volume.

Parameters

- **volume** (*float*) – The volume to aspirate.
- **flowRate** (*float*) – The desired flow rate.
- **waitUntilDone** (*bool*) – Whether to block program execution until calibration is completed.
- **switchValveWhenDone** (*bool*) – If *True*, switch the valve to dispense position after the aspiration is finished. Implies *wait_until_done=True*.

```
calibrate(waitUntilDone=False)
```

Calibrate the syringe pump.

You must not use this function if a syringe is installed in the pump as the syringe may be damaged!

Parameters **waitUntilDone** (*bool*) – Whether to block program execution until calibration is completed.

```
clearFaultState()
```

Switch the pump back to an operational state after an error had occurred.

```
dispense(volume, flowRate, waitUntilDone=False, switchValveWhenDone=False)
```

Dispense the specified volume.

Parameters

- **volume** (*float*) – The volume to dispense.
- **flowRate** (*float*) – The desired flow rate.

- **waitUntilDone** (*bool*) – Whether to block program execution until calibration is completed.
- **switchValveWhenDone** (*bool*) – If *True*, switch the valve to aspiration position after the dispense is finished. Implies *wait_until_done=True*.

empty (*flowRate*, *waitUntilDone=False*, *switchValveWhenDone=False*)

Empty the syringe entirely.

Parameters

- **flowRate** (*float*) – The desired flow rate.
- **waitUntilDone** (*bool*) – Whether to block program execution until calibration is completed.
- **switchValveWhenDone** (*bool*) – If *True*, switch the valve to aspirate position after the dispensing is finished. Implies *wait_until_done=True*.

fill (*flowRate*, *waitUntilDone=False*, *switchValveWhenDone=False*)

Fill the syringe entirely.

Parameters

- **flowRate** (*float*) – The desired flow rate.
- **waitUntilDone** (*bool*) – Whether to block program execution until calibration is completed.
- **switchValveWhenDone** (*bool*) – If *True*, switch the valve to dispense position after the aspiration is finished. Implies *wait_until_done=True*.

property fillLevel

Current fill level of the syringe.

property flowRateUnit

The unit in which flow rates are provided.

property isInFaultState

Whether the pump is currently in a non-operational “fault state”.

To enable the pump again, call `clearFaultState()`.

property maxFlowRate

Maximum flow rate the pump can provide with the installed syringe.

stop()

Stop any pump operation immediately.

switchValvePosition()

Switch the valve to the opposite position.

property syringeType

Type of the installed syringe.

property volumeUnit

The unit in which the volumes are provided.

`psychopy.hardware.findPhotometer` (*ports=None*, *device=None*)

Try to find a connected photometer/photospectrometer!

PsychoPy will sweep a series of serial ports trying to open them. If a port successfully opens then it will try to issue a command to the device. If it responds with one of the expected values then it is assumed to be the appropriate device.

Parameters

ports [a list of ports to search] Each port can be a string (e.g. 'COM1', '/dev/tty.Keyspan1.1') or a number (for win32 comports only). If none are provided then PsychoPy will sweep COM0-10 on win32 and search known likely port names on macOS and Linux.

device [string giving expected device (e.g. 'PR650', 'PR655', 'CS100A', 'LS100', 'LS110', 'S470'). If this is not given then an attempt will be made to find a device of any type, but this often fails

Returns

- An object representing the first photometer found
- None if the ports didn't yield a valid response
- None if there were not even any valid ports (suggesting a driver not being installed)

e.g.:

```
# sweeps ports 0 to 10 searching for a PR655
photom = findPhotometer(device='PR655')
print(photom.getLum())
if hasattr(photom, 'getSpectrum'):
    # can retrieve spectrum (e.g. a PR650)
    print(photom.getSpectrum())
```

9.6 psychopy.iohub - ioHub event monitoring framework

ioHub monitors for device events in parallel with the experiment execution by running in a separate process than the main script. This means, for instance, that keyboard and mouse event timing is not quantized by the rate at which the `window.flip()` method is called.

ioHub reports device events to the experiment runtime as they occur. Optionally, events can be saved to a [HDF5](#) file.

All iohub events are timestamped using the global time base (`psychopy.core.getTime()`). Events can be accessed as a device independent event stream, or from a specific device of interest.

A comprehensive set of examples that each use at least one of the iohub devices is available in the `psychopy/demos/coder/iohub` folder.

9.6.1 Starting the psychopy.iohub Process

To use ioHub within your Coder experiment script, ioHub needs to be started at the beginning of the experiment script.

The easiest way to do this is by calling the `launchHubServer` function.

launchHubServer Function

`psychopy.iohub.client.launchHubServer (**kwargs)`

Starts the ioHub Server subprocess, and return a `psychopy.iohub.client.ioHubConnection` object that is used to access enabled iohub device's events, get events, and control the ioHub process during the experiment.

By default (no kwargs specified), the ioHub server does not create an ioHub HDF5 file, events are available to the experiment program at runtime. The following Devices are enabled by default:

- Keyboard: named 'keyboard', with runtime event reporting enabled.
- Mouse: named 'mouse', with runtime event reporting enabled.
- Monitor: named 'monitor'.
- Experiment: named 'experiment'.

To customize how the ioHub Server is initialized when started, use one or more of the following keyword arguments when calling the function:

Parameters

- **experiment_code** (*str*, <= 256 char) – If experiment_code is provided, an ioHub HDF5 file will be created for the session.
- **session_code** (*str*, <= 256 char) – When specified, used as the name of the ioHub HDF5 file created for the session.
- **experiment_info** (*dict*) – Can be used to save the following experiment metadata fields: code (<=256 chars), title (<=256 chars), description (<=4096 chars), version (<=32 chars)
- **session_info** (*dict*) – Can be used to save the following session metadata fields: code (<=256 chars), name (<=256 chars), comments (<=4096 chars), user_variables (dict)
- **datastore_name** (*str*) – Used to provide an ioHub HDF5 file name different than the session_code.
- **window** (*psychopy.visual.Window*) – The psychoPy experiment window being used. Information like display size, viewing distance, coord / color type is used to update the ioHub Display device.
- **iohub_config_name** (*str*) – Specifies the name of the iohub_config.yaml file that contains the ioHub Device list to be used by the ioHub Server. i.e. the 'device_list' section of the yaml file.
- **iohub.device.path** (*str*) – Add an ioHub Device by using the device class path as the key, and the device's configuration in a dict value.
- **psychopy_monitor** (*(deprecated)*) – The path to a Monitor Center config file
- **Examples** –

A. Wait for the 'q' key to be pressed:

```
from psychopy.iohub.client import launchHubServer

# Start the ioHub process. 'io' can now be used during the
# experiment to access iohub devices and read iohub device_
↪events.
io=launchHubServer()
```

(continues on next page)

(continued from previous page)

```

print("Press any Key to Exit Example.....")

# Wait until a keyboard event occurs
keys = io.devices.keyboard.waitForKeys(keys=['q',])

print("Key press detected: {}".format(keys))
print("Exiting experiment....")

# Stop the ioHub Server
io.quit()
    
```

- see the `psychopy/demos/coder/iohub/launchHub.py` demo for examples (Please) –
- different ways to use the `launchHubServer` function. (of) –

ioHubConnection Class

The `psychopy.iohub.ioHubConnection` object returned from the `launchHubServer` function provides methods for controlling the iohub process and accessing iohub devices and events.

class `psychopy.iohub.client.ioHubConnection` (*ioHubConfig=None*, *ioHubConfigAbsPath=None*)

`ioHubConnection` is responsible for creating, sending requests to, and reading replies from the ioHub Process. This class is also used to shut down and disconnect the ioHub Server process.

The `ioHubConnection` class is also used as the interface to any ioHub Device instances that have been created so that events from the device can be monitored. These device objects can be accessed via the `ioHubConnection.devices` attribute, providing ‘dot name’ access to enabled devices. Alternatively, the `.getDevice(name)` method can be used and will return `None` if the device name specified does not exist.

Using the `.devices` attribute is handy if you know the name of the device to be accessed and you are sure it is actually enabled on the ioHub Process.

An example of accessing a device using the `.devices` attribute:

```

# get the Mouse device, named mouse
mouse=hub.devices.mouse
mouse_position = mouse.getPosition()

print 'mouse position: ', mouse_position

# Returns something like:
# >> mouse position: [-211.0, 371.0]
    
```

getDevice (*deviceName*)

Returns the `ioHubDeviceView` that has a matching name (based on the `device : name` property specified in the `iohub_config.yaml` for the experiment). If no device with the given name is found, `None` is returned. Example, accessing a Keyboard device that was named ‘kb’

```

keyboard = self.getDevice('kb')
kb_events= keyboard.getEvent()
    
```

This is the same as using the ‘natural naming’ approach supported by the `.devices` attribute, i.e:

```

keyboard = self.devices.kb
kb_events= keyboard.getEvent()
    
```

However the advantage of using `getDevice(device_name)` is that an exception is not created if you provide an invalid device name, or if the device is not enabled on the ioHub server; `None` is returned instead.

Parameters `deviceName` (*str*) – Name given to the ioHub Device to be returned

Returns The `ioHubDeviceView` instance for `deviceName`.

getEvents (*device_label=None, as_type='namedtuple'*)

Retrieve any events that have been collected by the ioHub Process from monitored devices since the last call to `getEvents()` or `clearEvents()`.

By default all events for all monitored devices are returned, with each event being represented as a named-tuple of all event attributes.

When events are retrieved from an event buffer, they are removed from that buffer as well.

If events are only needed from one device instead of all devices, providing a valid device name as the `device_label` argument will result in only events from that device being returned.

Events can be received in one of several object types by providing the optional `as_type` property to the method. Valid values for `as_type` are the following str values:

- `'list'`: Each event is a list of ordered attributes.
- `'namedtuple'`: Each event is converted to a namedtuple object.
- `'dict'`: Each event converted to a dict object.
- **`'object'`: Each event is converted to a DeviceEvent subclass** based on the event's type.

Parameters

- **`device_label`** (*str*) – Name of device to retrieve events for. If `None` (the default) returns device events from all devices.
- **`as_type`** (*str*) – Returned event object type. Default: `'namedtuple'`.

Returns List of event objects; object type controlled by `'as_type'`.

Return type `tuple`

clearEvents (*device_label='all'*)

Clears unread events from the ioHub Server's Event Buffer(s) so that unneeded events are not discarded.

If `device_label` is `'all'`, (the default), then events from both the ioHub *Global Event Buffer* and all *Device Event Buffer's* are cleared.

If `device_label` is `None` then all events in the ioHub *Global Event Buffer* are cleared, but the *Device Event Buffers* are unaffected.

If `device_label` is a str giving a valid device name, then that *Device Event Buffer* is cleared, but the *Global Event Buffer* is not affected.

Parameters `device_label` (*str*) – device name, `'all'`, or `None`

Returns `None`

sendMessageEvent (*text, category="", offset=0.0, sec_time=None*)

Create and send an Experiment MessageEvent to the ioHub Server for storage in the ioDataStore hdf5 file.

Parameters

- **`text`** (*str*) – The text message for the message event. 128 char max.
- **`category`** (*str*) – A str grouping code for the message. Optional. 32 char max.

- **offset** (*float*) – Optional sec.msec offset applied to the message event time stamp. Default 0.
- **sec_time** (*float*) – Absolute sec.msec time stamp for the message in. If not provided, or None, then the MessageEvent is time stamped when this method is called using the global timer (core.getTime()).

cacheMessageEvent (*text, category="", offset=0.0, sec_time=None*)

Create an Experiment MessageEvent and store in local cache. Message must be sent before it is saved to hdf5 file.

Parameters

- **text** (*str*) – The text message for the message event. 128 char max.
- **category** (*str*) – A str grouping code for the message. Optional. 32 char max.
- **offset** (*float*) – Optional sec.msec offset applied to the message event time stamp. Default 0.
- **sec_time** (*float*) – Absolute sec.msec time stamp for the message in. If not provided, or None, then the MessageEvent is time stamped when this method is called using the global timer (core.getTime()).

createTrialHandlerRecordTable (*trials, cv_order=None*)

Create a condition variable table in the ioHub data file based on the a psychopy TrialHandler. By doing so, the ioHub data file can contain the DV and IV values used for each trial of an experiment session, along with all the ioHub device events recorded by ioHub during the session.

Example psychopy code usage:

```
# Load a trial handler and
# create an associated table in the ioHub data file
#
from psychopy.data import TrialHandler, importConditions

exp_conditions=importConditions('trial_conditions.xlsx')
trials = TrialHandler(exp_conditions, 1)

# Inform the ioHub server about the TrialHandler
#
io.createTrialHandlerRecordTable(trials)

# Read a row of the trial handler for
# each trial of your experiment
#
for trial in trials:
    # do whatever...

# During the trial, trial variable values can be updated
#
trial['TRIAL_START']=flip_time

# At the end of each trial, before getting
# the next trial handler row, send the trial
# variable states to ioHub so they can be stored for future
# reference.
#
io.addTrialHandlerRecord(trial)
```

addTrialHandlerRecord (*cv_row*)

Adds the values from a TrialHandler row / record to the iohub data file for future data analysis use.

Parameters *cv_row* –

Returns None

getTime ()

Deprecated Method: Use Computer.getTime instead. Remains here for testing time bases between processes only.

setPriority (*level='normal', disable_gc=False*)

See Computer.setPriority documentation, where current process will be the iohub process.

getPriority ()

See Computer.getPriority documentation, where current process will be the iohub process.

getProcessAffinity ()

Returns the current **ioHub Process** affinity setting, as a list of ‘processor’ id’s (from 0 to getSystemProcessorCount()-1). A Process’s Affinity determines which CPU’s or CPU cores a process can run on. By default the ioHub Process can run on any CPU or CPU core.

This method is not supported on OS X at this time.

Parameters None –

Returns

A list of integer values between 0 and Computer.getSystemProcessorCount()-1, where values in the list indicate processing unit indexes that the ioHub process is able to run on.

Return type list

setProcessAffinity (*processor_list*)

Sets the **ioHub Process** Affinity based on the value of processor_list.

A Process’s Affinity determines which CPU’s or CPU cores a process can run on. By default the ioHub Process can run on any CPU or CPU core.

The processor_list argument must be a list of ‘processor’ id’s; integers in the range of 0 to Computer.processing_unit_count-1, representing the processing unit indexes that the ioHub Server should be allowed to run on.

If processor_list is given as an empty list, the ioHub Process will be able to run on any processing unit on the computer.

This method is not supported on OS X at this time.

Parameters *processor_list* (*list*) – A list of integer values between 0 and Computer.processing_unit_count-1, where values in the list indicate processing unit indexes that the ioHub process is able to run on.

Returns None

flushDataStoreFile ()

Manually tell the iohub datastore to flush any events it has buffered in memory to disk. Any cached message events are sent to the iohub server before flushing the iohub datastore.

Parameters None –

Returns None

startCustomTasklet (*task_name, task_class_path, **class_kwargs*)

Instruct the iohub server to start running a custom tasklet given by task_class_path. It is important that the

custom task does not block for any significant amount of time, or the processing of events by the iohub server will be negatively effected.

See the customtask.py demo for an example of how to make a long running task not block the rest of the iohub server.

stopCustomTasklet (*task_name*)

Instruct the iohub server to stop the custom task that was previously started by calling self.startCustomTasklet(...). task_name identifies which custom task should be stopped and must match the task_name of a previously started custom task.

shutdown ()

Tells the ioHub Server to close all ioHub Devices, the ioDataStore, and the connection monitor between the PsychoPy and ioHub Processes. Then end the server process itself.

Parameters None –

Returns None

quit ()

Same as the shutdown() method, but has same name as PsychoPy core.quit() so maybe easier to remember.

_startServer (*ioHubConfig=None, ioHubConfigAbsPath=None*)

Starts the ioHub Process, storing it's process id, and creating the experiment side device representation for IPC access to public device methods.

_createDeviceList (*monitor_devices_config*)

Create client side iohub device views.

_addDeviceView (*dev_cls_name, dev_config*)

Add an iohub device view to self.devices

_sendToHubServer (*tx_data*)

General purpose local <-> iohub server process UDP based request - reply code. The method blocks until the request is fulfilled and a response is received from the ioHub server.

Parameters **tx_data** (*tuple*) – data to send to iohub server

Return (object): response from the ioHub Server process.

_sendExperimentInfo (*experimentInfoDict*)

Sends the experiment info from the experiment config file to the ioHub Server, which passes it to the ioDataStore, determines if the experiment already exists in the hdf5 file based on 'experiment_code', and returns a new or existing experiment ID based on that criteria.

_sendSessionInfo (*sess_info*)

Sends the experiment session info from the experiment config file and the values entered into the session dialog to the ioHub Server, which passes it to the ioDataStore.

The dataStore determines if the session already exists in the experiment file based on 'session_code', and returns a new session ID if session_code is not in use by the experiment.

static _isErrorReply (*data*)

Check if an iohub server reply contains an error that should be raised by the local process.

9.6.2 Supported Devices

psychopy.iohub supports several different device types.

Details for each device can be found in the following sections.

Keyboard Device

The iohub Keyboard device provides methods to:

- Check for any new keyboard events that have occurred since the last time keyboard events were checked or cleared.
- Wait until a keyboard event occurs.
- Clear the device of any unread events.
- Get a list of all currently pressed keys.

class psychopy.iohub.client.keyboard.**Keyboard** (*ioclient, dev_cls_name, dev_config*)

The Keyboard device provides access to KeyboardPress and KeyboardRelease events as well as the current keyboard state.

Examples

A. Print all keyboard events received for 5 seconds:

```
from psychopy.iohub import launchHubServer
from psychopy.core import getTime

# Start the ioHub process. 'io' can now be used during the
# experiment to access iohub devices and read iohub device events.
io = launchHubServer()

keyboard = io.devices.keyboard

# Check for and print any Keyboard events received for 5 seconds.
stime = getTime()
while getTime()-stime < 5.0:
    for e in keyboard.getEvents():
        print(e)

# Stop the ioHub Server
io.quit()
```

B. Wait for a keyboard press event (max of 5 seconds):

```
from psychopy.iohub import launchHubServer
from psychopy.core import getTime

# Start the ioHub process. 'io' can now be used during the
# experiment to access iohub devices and read iohub device events.
io = launchHubServer()

keyboard = io.devices.keyboard

# Wait for a key keypress event ( max wait of 5 seconds )
```

(continues on next page)

(continued from previous page)

```

presses = keyboard.waitForPresses(maxWait=5.0)

print(presses)

# Stop the ioHub Server
io.quit()
    
```

`_syncDeviceState()`

An optimized iohub server request that receives all device state and event information in one response.

Returns None

`getKey` (*keys=None, chars=None, mods=None, duration=None, etype=None, clear=True*)

Return a list of any KeyboardPress or KeyboardRelease events that have occurred since the last time either:

- this method was called with the kwarg `clear=True` (default)
- the `keyboard.clear()` method was called.

Other than the ‘clear’ kwarg, any kwargs that are not None or an empty list are used to filter the possible events that can be returned. If multiple filter criteria are provided, only events that match **all** specified criteria are returned.

If no KeyboardEvent’s are found that match the filtering criteria, an empty tuple is returned.

Returned events are sorted by time.

Parameters

- **keys** – Include events where `.key` in `keys`.
- **chars** – Include events where `.char` in `chars`.
- **mods** – Include events where `.modifiers` include ≥ 1 mods element.
- **duration** – Include KeyboardRelease events where `.duration > duration` or `.duration < -(duration)`.
- **etype** – Include events that match `etype` of `Keyboard.KEY_PRESS` or `Keyboard.KEY_RELEASE`.
- **clear** – True (default) = clear returned events from event buffer, False = leave the keyboard event buffer unchanged.

Returns tuple of KeyboardEvent instances, or ()

`getPresses` (*keys=None, chars=None, mods=None, clear=True*)

See the `getKey()` method documentation.

This method is identical, but only returns KeyboardPress events.

`getReleases` (*keys=None, chars=None, mods=None, duration=None, clear=True*)

See the `getKey()` method documentation.

This method is identical, but only returns KeyboardRelease events.

property reporting

Specifies if the the keyboard device is reporting / recording events.

- True: keyboard events are being reported.
- False: keyboard events are not being reported.

By default, the Keyboard starts reporting events automatically when the ioHub process is started and continues to do so until the process is stopped.

This property can be used to read or set the device reporting state:

```
# Read the reporting state of the keyboard.
is_reporting_keyboard_event = keyboard.reporting

# Stop the keyboard from reporting any new events.
keyboard.reporting = False
```

property state

time values. The key is taken from the originating press event .key field. The time value is time of the key press event.

Note that any pressed, or active, modifier keys are included in the return value.

Returns dict

Type Returns all currently pressed keys as a dictionary of key

waitForKeys (*maxWait=None, keys=None, chars=None, mods=None, duration=None, etype=None, clear=True, checkInterval=0.002*)

Blocks experiment execution until at least one matching KeyboardEvent occurs, or until maxWait seconds has passed since the method was called.

Keyboard events are filtered the same way as in the getKeys() method.

As soon as at least one matching KeyboardEvent occurs prior to maxWait, the matching events are returned as a tuple.

Returned events are sorted by time.

Parameters

- **maxWait** – Maximum seconds method waits for ≥ 1 matching event. If ≤ 0.0 , method functions the same as getKeys(). If None, the methods blocks indefinitely.
- **keys** – Include events where .key in keys.
- **chars** – Include events where .char in chars.
- **mods** – Include events where .modifiers include ≥ 1 mods element.
- **duration** – Include KeyboardRelease events where .duration > duration or .duration < -(duration).
- **etype** – Include events that match etype of Keyboard.KEY_PRESS or Keyboard.KEY_RELEASE.
- **clear** – True (default) = clear returned events from event buffer, False = leave the keyboard event buffer unchanged.
- **checkInterval** – The time between getKeys() calls while waiting. The method sleeps between getKeys() calls, up until checkInterval*2.0 sec prior to the maxWait. After that time, keyboard events are constantly checked until the method times out.

Returns tuple of KeyboardEvent instances, or ()

waitForPresses (*maxWait=None, keys=None, chars=None, mods=None, duration=None, clear=True, checkInterval=0.002*)

See the waitForKeys() method documentation.

This method is identical, but only returns KeyboardPress events.

waitForReleases (*maxWait=None, keys=None, chars=None, mods=None, duration=None, clear=True, checkInterval=0.002*)

See the `waitForKeys()` method documentation.

This method is identical, but only returns `KeyboardRelease` events.

Keyboard Events

The Keyboard device can return two types of events, which represent key press and key release actions on the keyboard.

KeyboardPress Event

class `psychopy.iohub.client.keyboard.KeyboardPress` (*ioe_array*)

An iohub Keyboard device key press event.

property char

The unicode value of the keyboard event, if available. This field is only populated when the keyboard event results in a character that could be printable.

Returns unicode, "" if no char value is available for the event.

property device

The `ioHubDeviceView` that is associated with the event, i.e. the iohub device view for the device that generated the event.

Returns `ioHubDeviceView`

property modifiers

A list of any modifier keys that were pressed when this keyboard event occurred. Each element of the list contains a keyboard modifier string constant. Possible values are:

- 'lctrl', 'rctrl'
- 'lshift', 'rshift'
- 'alt', 'ralt' (labelled as 'option' keys on Apple Keyboards)
- 'lcmd', 'rcmd' (map to the 'windows' key(s) on Windows keyboards)
- 'menu'
- 'capslock'
- 'numlock'
- 'function' (OS X only)
- 'modhelp' (OS X only)

If no modifiers were active when the event occurred, an empty list is returned.

Returns tuple

property time

The time stamp of the event. Uses the same time base that is used by `psychopy.core.getTime()`

Returns float

property type

The event type string constant.

Returns str

KeyboardRelease Event

class psychopy.iohub.client.keyboard.**KeyboardRelease** (*ioe_array*)

An iohub Keyboard device key release event.

property duration

The duration (in seconds) of the key press. This is calculated by subtracting the current event.time from the associated keypress.time.

If no matching keypress event was reported prior to this event, then 0.0 is returned. This can happen, for example, when the key was pressed prior to psychopy starting to monitor the device. This condition can also happen when keyboard.reset() method is called between the press and release event times.

Returns float

property pressEventID

The event.id of the associated press event.

The key press id is 0 if no associated KeyboardPress event was found. See the duration property documentation for details on when this can occur.

Returns unsigned int

property char

The unicode value of the keyboard event, if available. This field is only populated when the keyboard event results in a character that could be printable.

Returns unicode, "" if no char value is available for the event.

property device

The ioHubDeviceView that is associated with the event, i.e. the iohub device view for the device that generated the event.

Returns ioHubDeviceView

property modifiers

A list of any modifier keys that were pressed when this keyboard event occurred. Each element of the list contains a keyboard modifier string constant. Possible values are:

- 'lctrl', 'rctrl'
- 'lshift', 'rshift'
- 'alt', 'ralt' (labelled as 'option' keys on Apple Keyboards)
- 'lcmd', 'rcmd' (map to the 'windows' key(s) on Windows keyboards)
- 'menu'
- 'capslock'
- 'numlock'
- 'function' (OS X only)
- 'modhelp' (OS X only)

If no modifiers were active when the event occurred, an empty list is returned.

Returns tuple

property time

The time stamp of the event. Uses the same time base that is used by psychopy.core.getTime()

Returns float

property type

The event type string constant.

Returns str

The ioHub Mouse Device

Platforms: Windows, macOS, Linux

Mouse Event Types

The Mouse device supports the following event types. Device events returned by `getEvents()` are automatically converted to either namedtuple or dictionary objects with the same attributes / keys as the associated event class attributes.

ioHub Common Eye Tracker Interface

The iohub common eye tracker interface provides a consistent way to configure and collected data from several different eye tracker manufacturers.

Supported Eye Trackers

The following eye trackers are currently supported by iohub.

Gazepoint

Platforms:

- Windows 7 / 10 only

Required Python Version:

- Python 3.6 +

Supported Models:

- Gazepoint GP3

Additional Software Requirements

To use your Gazepoint GP3 during an experiment you must first start the Gazepoint Control software on the computer running .

EyeTracker Class

class psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3.**EyeTracker**

To start iohub with a Gazepoint GP3 eye tracker device, add a GP3 device to the device dictionary passed to launchHubServer or the experiment's iohub_config.yaml:

```
eyetracker.hw.gazepoint.gp3.EyeTracker
```

Note: The Gazepoint control application **must** be running while using this interface.

Examples

A. Start ioHub with Gazepoint GP3 device and run tracker calibration:

```
from psychopy.iohub import launchHubServer
from psychopy.core import getTime, wait

iohub_config = {'eyetracker.hw.gazepoint.gp3.EyeTracker':
                {'name': 'tracker', 'device_timer': {'interval': 0.005}}}

io = launchHubServer(**iohub_config)

# Get the eye tracker device.
tracker = io.devices.tracker

# run eyetracker calibration
r = tracker.runSetupProcedure()
```

B. Print all eye tracker events received for 2 seconds:

```
# Check for and print any eye tracker events received...
tracker.setRecordingState(True)

stime = getTime()
while getTime()-stime < 2.0:
    for e in tracker.getEvents():
        print(e)
```

C. Print current eye position for 5 seconds:

```
# Check for and print current eye position every 100 msec.
stime = getTime()
while getTime()-stime < 5.0:
    print(tracker.getPosition())
    wait(0.1)

tracker.setRecordingState(False)

# Stop the ioHub Server
io.quit()
```

clearEvents (*event_type=None, filter_id=None, call_proc_events=True*)

Clears any DeviceEvents that have occurred since the last call to the device's getEvents(), or clearEvents() methods.

Note that calling `clearEvents()` at the device level only clears the given device's event buffer. The ioHub Process's Global Event Buffer is unchanged.

Parameters `None` –

Returns `None`

enableEventReporting (*enabled=True*)

`enableEventReporting` is functionally identical to the eye tracker device specific `setRecordingState` method.

getConfiguration ()

Retrieve the configuration settings information used to create the device instance. This will be the default settings for the device, found in `iohub.devices.<device_name>.default_<device_name>.yaml`, updated with any device settings provided via `launchHubServer(...)`.

Changing any values in the returned dictionary has no effect on the device state.

Parameters `None` –

Returns The dictionary of the device configuration settings used to create the device.

Return type (`dict`)

getEvents (*args, **kwargs)

Retrieve any `DeviceEvents` that have occurred since the last call to the device's `getEvents()` or `clearEvents()` methods.

Note that calling `getEvents()` at a device level does not change the Global Event Buffer's contents.

Parameters

- **event_type_id** (*int*) – If specified, provides the ioHub `DeviceEvent` ID for which events should be returned for. Events that have occurred but do not match the event ID specified are ignored. Event type ID's can be accessed via the `EventConstants` class; all available event types are class attributes of `EventConstants`.
- **clearEvents** (*int*) – Can be used to indicate if the events being returned should also be removed from the device event buffer. `True` (the default) indicates to remove events being returned. `False` results in events being left in the device event buffer.
- **asType** (*str*) – Optional kwarg giving the object type to return events as. Valid values are 'namedtuple' (the default), 'dict', 'list', or 'object'.

Returns New events that the ioHub has received since the last `getEvents()` or `clearEvents()` call to the device. Events are ordered by the ioHub time of each event, older event at index 0. The event object type is determined by the `asType` parameter passed to the method. By default a `namedtuple` object is returned for each event.

Return type (`list`)

getLastGazePosition ()

The `getLastGazePosition` method returns the most recent eye gaze position received from the Eye Tracker. This is the position on the calibrated 2D surface that the eye tracker is reporting as the current eye position. The units are in the units in use by the ioHub Display device.

If binocular recording is being performed, the average position of both eyes is returned.

If no samples have been received from the eye tracker, or the eye tracker is not currently recording data, `None` is returned.

Parameters `None` –

Returns

If this method is not supported by the eye tracker interface, `EyeTrackerConstants.EYETRACKER_INTERFACE_METHOD_NOT_SUPPORTED` is returned.

None: If the eye tracker is not currently recording data or no eye samples have been received.

tuple: Latest (`gaze_x`, `gaze_y`) position of the eye(s)

Return type `int`

getLastSample()

The `getLastSample` method returns the most recent eye sample received from the Eye Tracker. The Eye Tracker must be in a recording state for a sample event to be returned, otherwise None is returned.

Parameters None –

Returns

If this method is not supported by the eye tracker interface, `EyeTrackerConstants.FUNCTIONALITY_NOT_SUPPORTED` is returned.

None: If the eye tracker is not currently recording data.

`EyeSample`: If the eye tracker is recording in a monocular tracking mode, the latest sample event of this event type is returned.

`BinocularEyeSample`: If the eye tracker is recording in a binocular tracking mode, the latest sample event of this event type is returned.

Return type `int`

getPosition()

See `getLastGazePosition()`.

isRecordingEnabled()

`isRecordingEnabled` returns the recording state from the eye tracking device.

Returns `True` == the device is recording data; `False` == Recording is not occurring

Return type `bool`

runSetupProcedure(calibration_args={})

Start the eye tracker calibration procedure.

setRecordingState(recording)

`setRecordingState` is used to start or stop the recording of data from the eye tracking device.

Parameters `recording` (`bool`) – if `True`, the eye tracker will start recording available eye data and sending it to the experiment program if data streaming was enabled for the device.

If `recording` == `False`, then the eye tracker stops recording eye data and streaming it to the experiment.

If the eye tracker is already recording, and `setRecordingState(True)` is called, the eye tracker will simply continue recording and the method call is a no-op. Likewise if the system has already stopped recording and `setRecordingState(False)` is called again.

Parameters `recording` (`bool`) – if `True`, the eye tracker will start recording data.; `false` = stop recording data.

Return:trackerTime `bool`: the current recording state of the eye tracking device

trackerSec()

Same as the GP3 implementation of `trackerTime()`.

trackerTime ()

Current eye tracker time in the eye tracker’s native time base. The GP3 system uses a sec.usec timebase based on the Windows QPC, so when running on a single computer setup, iohub can directly read the current gaze point time. When running with a two computer setup, current gaze point time is assumed to equal current local time.

Returns current native eye tracker time in sec.msec format.

Return type float

Supported Event Types

The Gaze point GP3 provides real-time access to binocular sample data. iohub creates a BinocularEyeSampleEvent for each sample received from the GP3.

The following fields of the BinocularEyeSample event are supported:

class psychopy.iohub.devices.eyetracker.**BinocularEyeSampleEvent** (*args, **kwargs)

The BinocularEyeSampleEvent event represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recording both eyes of a participant.

Event Type ID: EventConstants.BINOCULAR_EYE_SAMPLE

Event Type String: ‘BINOCULAR_EYE_SAMPLE’

time

time of event, in sec.msec format, using psychopy timebase.

left_gaze_x

The horizontal position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gaze point LPOGX field.

left_gaze_y

The vertical position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gaze point LPOGY field.

left_raw_x

The uncalibrated x position of the left eye in a device specific coordinate space. Uses Gaze point LPCX field.

left_raw_y

The uncalibrated y position of the left eye in a device specific coordinate space. Uses Gaze point LPCY field.

left_pupil_measure_1

Left eye pupil diameter. (in camera pixels??). Uses Gaze point LPD field.

right_gaze_x

The horizontal position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gaze point RPOGX field.

right_gaze_y

The vertical position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gaze point RPOGY field.

right_raw_x

The uncalibrated x position of the right eye in a device specific coordinate space. Uses Gaze point RPCX field.

right_raw_y

The uncalibrated y position of the right eye in a device specific coordinate space. Uses Gazepoint RPCY field.

right_pupil_measure_1

Right eye pupil diameter. (in camera pixels??). Uses Gazepoint RPD field.

status

Indicates if eye sample contains 'valid' data for left and right eyes. 0 = Eye sample is OK. 2 = Right eye data is likely invalid. 20 = Left eye data is likely invalid. 22 = Eye sample is likely invalid.

iohub also creates basic start and end fixation events by using Gazepoint FPOG* fields. Identical / duplicate fixation events are created for the left and right eye.

class psychopy.iohub.devices.eyetracker.**FixationStartEvent** (*args, **kwargs)

A FixationStartEvent is generated when the beginning of an eye fixation (in very general terms, a period of relatively stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: EventConstants.FIXATION_START

Event Type String: 'FIXATION_START'

time

time of event, in sec.msec format, using psychopy timebase.

eye

Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

gaze_x

The calibrated horizontal eye position on the computer screen at the start of the fixation. Units are same as Display. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazepoint FPOGX field.

gaze_y

The calibrated horizontal eye position on the computer screen at the start of the fixation. Units are same as Display. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazepoint FPOGY field.

class psychopy.iohub.devices.eyetracker.**FixationEndEvent** (*args, **kwargs)

A FixationEndEvent is generated when the end of an eye fixation (in very general terms, a period of relatively stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: EventConstants.FIXATION_END

Event Type String: 'FIXATION_END'

time

time of event, in sec.msec format, using psychopy timebase.

eye

Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

average_gaze_x

Average calibrated horizontal eye position during the fixation, specified in Display Units. Uses Gazepoint FPOGX field.

average_gaze_y

Average calibrated vertical eye position during the fixation, specified in Display Units. Uses Gazepoint FPOGY field.

duration

Duration of the fixation in sec.msec format. Uses Gazepoint FPOGD field.

Default Device Settings

```

eyetracker.hw.gazepoint.gp3.EyeTracker:
    # Indicates if the device should actually be loaded at experiment runtime.
    enable: True

    # The variable name of the device that will be used to access the ioHub Device_
    ↪class
    # during experiment run-time, via the devices.[name] attribute of the ioHub
    # connection or experiment runtime class.
    name: tracker

    # Should eye tracker events be saved to the ioHub DataStore file when the device
    # is recording data ?
    save_events: True

    # Should eye tracker events be sent to the Experiment process when the device
    # is recording data ?
    stream_events: True

    # How many eye events (including samples) should be saved in the ioHub event_
    ↪buffer before
    # old eye events start being replaced by new events. When the event buffer reaches
    # the maximum event length of the buffer defined here, older events will start to_
    ↪be dropped.
    event_buffer_length: 1024

    # The GP3 implementation of the common eye tracker interface supports the
    # BinocularEyeSampleEvent event type.
    monitor_event_types: [ BinocularEyeSampleEvent, FixationStartEvent,
    ↪FixationEndEvent]

    device_timer:
        interval: 0.005

    calibration:
        # target_duration is the number of sec.msec that a calibration point should
        # be displayed before moving onto the next point.
        # (Sets the GP3 CALIBRATE_TIMEOUT)
        target_duration: 1.25
        # target_delay specifies the target animation duration in sec.msec.
        # (Sets the GP3 CALIBRATE_DELAY)
        target_delay: 0.5

    # The model name of the device.
    model_name: GP3

    # The serial number of the GP3 device.
    serial_number:

    # manufacturer_name is used to store the name of the maker of the eye tracking
    # device. This is for informational purposes only.
    manufacturer_name: GazePoint
    
```

Last Updated: January, 2021

Pupil Labs - Core

Table of Contents

- *Pupil Labs - Core*
 - *High Level Pupil Core Introduction*
 - *Device, Software, and Connection Setup*
 - * *Additional Software Requirements*
 - * *Setting Up the Eye Tracker*
 - * *Setting Up*
 - * *Pupillometry + Gaze Mode*
 - *Implementation and API Overview*
 - * *EyeTracker Class*
 - * *Supported Event Types*
 - *Default Device Settings*

High Level Pupil Core Introduction

Pupil Core is a wearable eye tracker. The system consists of two inward-facing eye cameras and one forward-facing world camera mounted on a wearable eyeglasses-like frame.

Pupil Core provides gaze data in its world camera's field of view, regardless of the wearer's head position. As such, gaze can be analysed with the wearer looking and moving freely in their environment.

Pupil Core differs from remote eye trackers often used with . Remote eye trackers employ cameras mounted on or near a computer monitor. They provide gaze in screen-based coordinates, and this facilitates closed-loop analyses of gaze based on the known position of stimuli on-screen and eye gaze direction.

In order to use Pupil Core for screen-based work in , the screen will need to be robustly located within the world camera's field of view, and Pupil Core's gaze data subsequently transformed from world camera-based coordinates to screen-based coordinates. This is achieved with the use of [AprilTag Markers](#).



For a detailed overview of wearable *vs* remote eye trackers, check out [this Pupil Labs blog post](#).
Join the Pupil Labs [Discord community](#) to share your research and/or questions.

Device, Software, and Connection Setup

Additional Software Requirements

[Pupil Capture](#) version v2.0 or newer

Platforms:

- Windows 10
- macOS 10.14 or newer
- Ubuntu 16.04 or newer

Supported Models:

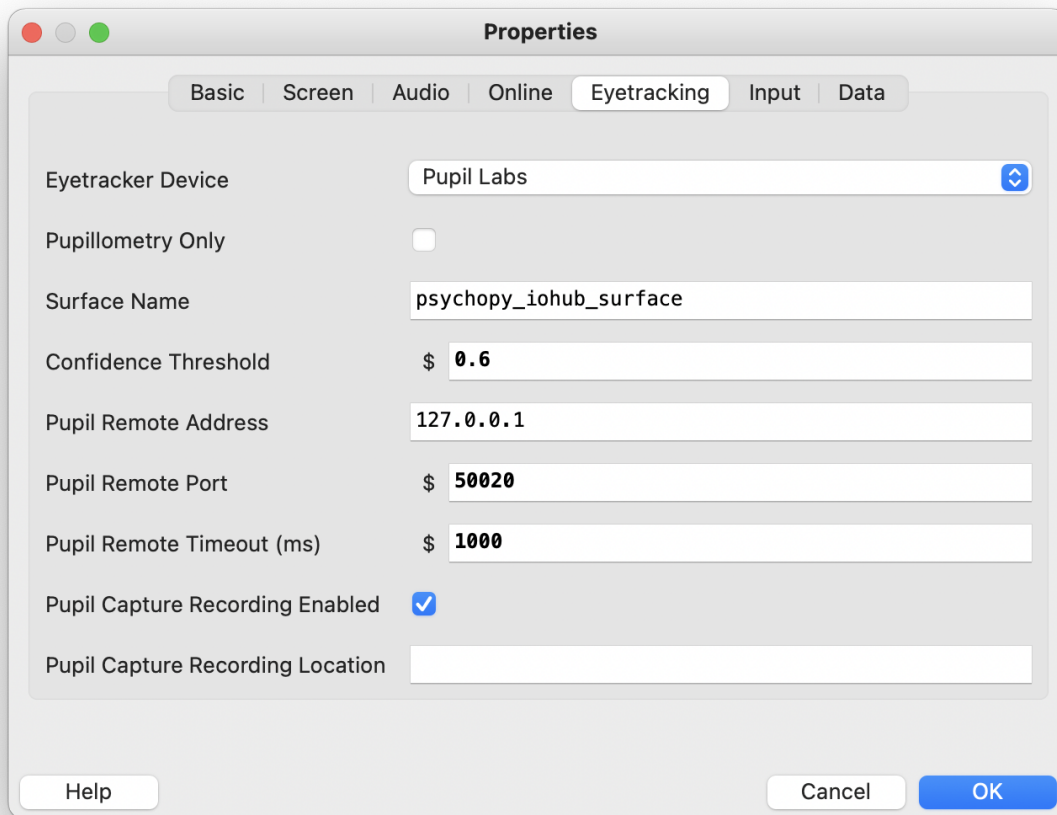
- Pupil Core headset

Setting Up the Eye Tracker

1. Follow [Pupil Core's Getting Started guide](#) to setup the headset and Capture software

Setting Up

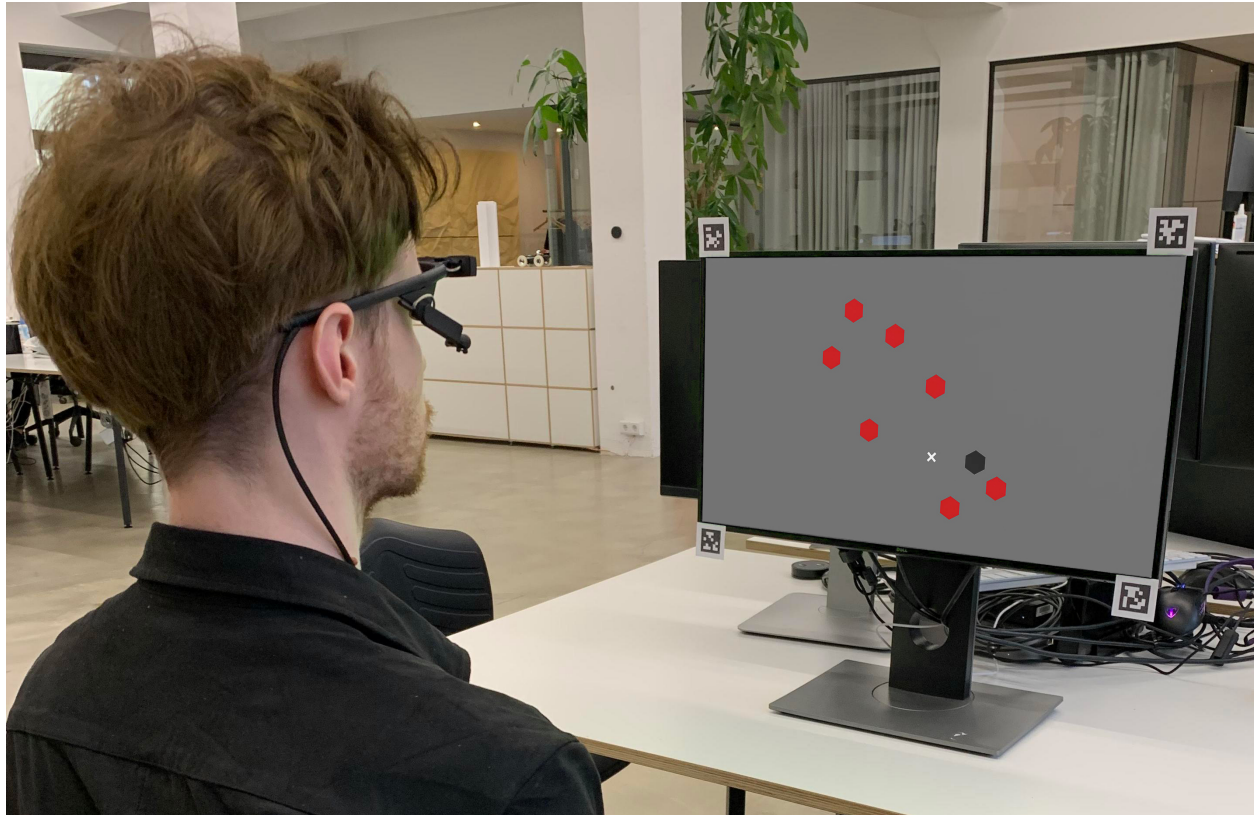
1. Open `experiment settings` in the Builder Window (cog icon in top panel)
2. Open the `Eyetracking` tab
3. Modify the properties as follows:
 - Select `Pupil Labs` from the `Eyetracker Device` drop down menu
 - `Pupil Remote Address / Port` - Defines how to connect to Pupil Capture. See Pupil Capture's *Network API* menu to check address and port are correct. will wait the amount of milliseconds declared in `Pupil Remote Timeout (ms)` for the connection to be established. An error will be raised if the timeout is reached.
 - `Pupil Capture Recording` - Enable this option to tell Pupil Capture to record the eye tracker's raw data during the experiment. You can read more about that in [Pupil Capture's official documentation](#). Leave `Pupil Capture Recording Location` empty to record to the default
 - `Gaze Confidence Threshold` - Set the minimum data quality received from Pupil Capture. Ranges from 0.0 (all data) to 1.0 (highest possible quality). We recommend using the default value of 0.6.
 - `Pupillometry Only` - If this mode is selected you will only receive pupillometry data. No further setup is required. If you are interested in gaze data, keep this option disabled and read on below.



Pupillometry + Gaze Mode

To receive gaze, enable Pupil Capture's Surface Tracking plugin:

1. Start by [printing four apriltag markers](#) and attaching them at the screen corners. Avoid occluding the screen and leave sufficient white space around the marker squares. Read more about the [general marker setup here](#).



1. Enable the surface tracker plugin
2. Define a surface and align its surface corners with the screen corners as good as possible
3. Rename the surface to the name set in the Surface Name field of the eye tracking project settings (default: `psychoPy_iohub_surface`)
4. Run the calibration component as part of your experiment

Implementation and API Overview

EyeTracker Class

class `psychoPy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core.EyeTracker` (*args, **kwargs)

Bases: `psychoPy.iohub.devices.eyetracker.EyeTrackerDevice`

Implementation of the Common Eye Tracker Interface for the Pupil Core headset.

Uses ioHub's polling method to process data from Pupil Capture's Network API.

To synchronize time between Pupil Capture and PsychoPy, the integration estimates the offset between their clocks and applies it to the incoming data. This step effectively transforms time between the two softwares while taking the transmission delay into account. For details, see this [real-time time-sync tutorial](#).

This class operates in two modes, depending on the `pupillometry_only` runtime setting:

1. **Pupillometry-only mode** If the `pupillometry_only` setting is to `True`, the integration will only receive eye-camera based metrics, e.g. pupil size, its location in eye camera coordinates, etc. The advantage of this mode is that it does not require calibrating the eye tracker or setting up AprilTag markers

for the AoI tracking. To receive gaze data in PsychoPy screen coordinates, see the Pupillometry+Gaze mode below.

Internally, this is implemented by subscribing to the `pupil.` data topic.

2. **Pupillometry+Gaze mode** If the `Pupillometry` `only` setting is set to `False`, the integration will receive positional data in addition to the pupillometry data mentioned above. For this to work, one has to setup Pupil Capture's built-in AoI tracking system and perform a calibration for each subject.

The integration takes care of translating the spatial coordinates to PsychoPy display coordinates.

Internally, this mode is implemented by subscribing to the `gaze.3d.` and the corresponding surface name data topics.`only`

Note: Only **one** instance of EyeTracker can be created within an experiment. Attempting to create > 1 instance will raise an exception.

getLastGazePosition () → Optional[Tuple[float, float]]

The `getLastGazePosition` method returns the most recent eye gaze position received from the Eye Tracker. This is the position on the calibrated 2D surface that the eye tracker is reporting as the current eye position. The units are in the units in use by the ioHub Display device.

If binocular recording is being performed, the average position of both eyes is returned.

If no samples have been received from the eye tracker, or the eye tracker is not currently recording data, `None` is returned.

Returns

- **None:** If the eye tracker is not currently recording data or no eye samples have been received.
- **tuple:** Latest (`gaze_x,gaze_y`) position of the eye(s)

getLastSample () → Union[None, *psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent*, *psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent*]

The `getLastSample` method returns the most recent eye sample received from the Eye Tracker. The Eye Tracker must be in a recording state for a sample event to be returned, otherwise `None` is returned.

Returns

- **MonocularEyeSampleEvent:** Gaze mapping result from a single pupil detection. Only emitted if a second eye camera is not being operated or the confidence of the pupil detection was insufficient for a binocular pair. See also this high-level overview of the [Pupil Capture Data Matching algorithm](#)
- **BinocularEyeSample:** Gaze mapping result from two combined pupil detections
- **None:** If the eye tracker is not currently recording data.

isConnected () → bool

`isConnected` returns whether the ioHub EyeTracker Device is connected to Pupil Capture or not. A Pupil Core headset must be connected and working properly for any of the Common Eye Tracker Interface functionality to work.

Parameters None –

Returns bool: True = the eye tracking hardware is connected. False otherwise.

isRecordingEnabled () → bool

The `isRecordingEnabled` method indicates if the eye tracker device is currently recording data.

Returns True == the device is recording data; False == Recording is not occurring

runSetupProcedure (*calibration_args*: *Optional[Dict] = None*) → *int*

The runSetupProcedure method starts the Pupil Capture calibration choreography.

Note: This is a blocking call for the PsychoPy Process and will not return to the experiment script until the calibration procedure was either successful, aborted, or failed.

Parameters **calibration_args** – This argument will be ignored and has only been added for the purpose of compatibility with the Common Eye Tracker Interface

Returns

- **EyeTrackerConstants.EYETRACKER_OK** if the calibration was successful
- **EyeTrackerConstants.EYETRACKER_SETUP_ABORTED** if the choreography was aborted by the user
- **EyeTrackerConstants.EYETRACKER_CALIBRATION_ERROR** if the calibration failed, check logs for details
- **EyeTrackerConstants.EYETRACKER_ERROR** if any other error occurred, check logs for details

setConnectionState (*enable*: *bool*) → *None*

setConnectionState either connects (setConnectionState(True)) or disables (setConnectionState(False)) active communication between the ioHub and Pupil Capture.

Note: A connection to the Eye Tracker is automatically established when the ioHub Process is initialized (based on the device settings in the iohub_config.yaml), so there is no need to explicitly call this method in the experiment script.

Note: Connecting an Eye Tracker to the ioHub does **not** necessarily collect and send eye sample data to the ioHub Process. To start actual data collection, use the Eye Tracker method setRecordingState(*bool*) or the ioHub Device method (device type independent) enableEventRecording(*bool*).

Parameters **enable** (*bool*) – True = enable the connection, False = disable the connection.

Returns *bool*: indicates the current connection state to the eye tracking hardware.

setRecordingState (*should_be_recording*: *bool*) → *bool*

The setRecordingState method is used to start or stop the recording and transmission of eye data from the eye tracking device to the ioHub Process.

If the pupil_capture_recording.enabled runtime setting is set to True, a corresponding raw recording within Pupil Capture will be started or stopped.

should_be_recording will also be passed to EyeTrackerDevice.enableEventReporting().

Parameters **recording** (*bool*) – if True, the eye tracker will start recording data.; false = stop recording data.

Returns *bool*: the current recording state of the eye tracking device

property surface_topic

Read-only Pupil Capture subscription topic to receive data from the configured surface

trackerSec() → float

Returns `EyeTracker.trackerTime()`

Returns The eye tracker hardware’s reported current time in sec.msec-usec format.

trackerTime() → float

Returns the current time reported by the eye tracker device.

Implementation measures the current time in PsychoPy time and applies the estimated clock offset to transform the measurement into tracker time.

Returns The eye tracker hardware’s reported current time.

Supported Event Types

The Pupil Core– integration provides real-time access to monocular and binocular sample data. In pupillometry-only mode, all events will be emitted as *MonocularEyeSampleEvents*. In pupillometry+gaze mode, the software only emits *BinocularEyeSampleEvents* events if Pupil Capture is driving a binocular headset and the detection from both eyes have sufficient *confidence* to be paired. See this high-level overview of the *Pupil Capture Data Matching algorithm* for details.

The supported fields are described below.

```
class psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent (*args,
                                                                **kwargs)
```

A *MonocularEyeSampleEvent* represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recoding from only one eye, or is recording from both eyes and averaging the binocular data.

Event Type ID: `EventConstants.MONOCULAR_EYE_SAMPLE`

Event Type String: ‘`MONOCULAR_EYE_SAMPLE`’

device_time: float

time of gaze measurement, in sec.msec format, using Pupil Capture clock

logged_time: float

time at which the sample was received in , in sec.msec format, using PsychoPy clock

time: float

time of gaze measurement, in sec.msec format, using PsychoPy clock

confidence_interval: float = -1.0

currently not supported, always set to -1.0

delay: float

The difference between `logged_time` and `time`, in sec.msec format

eye: int = 21 or 22

`psychopy.iohub.constants.EyeTrackerConstants.RIGHT_EYE` (22) or `psychopy.iohub.constants.EyeTrackerConstants.LEFT_EYE` (21)

gaze_x: float

x component of gaze location in display coordinates. Set to `float("nan")` in pupillometry-only mode.

gaze_y: float

y component of gaze location in display coordinates. Set to `float("nan")` in pupillometry-only mode.

gaze_z: `float = 0 or float("nan")`
 z component of gaze location in display coordinates. Set to `float("nan")` in pupillometry-only mode.
 Set to `0.0` otherwise.

eye_cam_x: `float`
 x component of 3d eye model location in undistorted eye camera coordinates

eye_cam_y: `float`
 y component of 3d eye model location in undistorted eye camera coordinates

eye_cam_z: `float`
 z component of 3d eye model location in undistorted eye camera coordinates

angle_x: `float`
 phi angle / horizontal rotation of the 3d eye model location in radians. $-\pi/2$ corresponds to looking directly into the eye camera

angle_y: `float`
 theta angle / vertical rotation of the 3d eye model location in radians. $\pi/2$ corresponds to looking directly into the eye camera

raw_x: `float`
 x component of the pupil center location in [normalized coordinates](#)

raw_y: `float`
 y component of the pupil center location in [normalized coordinates](#)

pupil_measure1: `float`
 Major axis of the detected pupil ellipse in pixels

pupil_measure1_type: `int = psychopy.iohub.constants.EyeTrackerConstants.PUPIL_MAJOR_AXIS`

pupil_measure2: `Optional[float]`
 Diameter of the detected pupil in mm or None if not available

pupil_measure2_type: `int = psychopy.iohub.constants.EyeTrackerConstants.PUPIL_DIAMETER`

class `psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent` (**args*,
***kwargs*)

The `BinocularEyeSampleEvent` event represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recording both eyes of a participant.

Event Type ID: `EventConstants.BINOCULAR_EYE_SAMPLE`

Event Type String: `'BINOCULAR_EYE_SAMPLE'`

device_time: `float`
 time of gaze measurement, in sec.msec format, using Pupil Capture clock

logged_time: `float`
 time at which the sample was received in PsychoPy, in sec.msec format, using PsychoPy clock

time: `float`
 time of gaze measurement, in sec.msec format, using PsychoPy clock

confidence_interval: `float = -1.0`
 currently not supported, always set to `-1.0`

delay: `float`
 The difference between `logged_time` and `time`, in sec.msec format

left_gaze_x: `float`
 x component of gaze location in display coordinates. Set to `float("nan")` in pupillometry-only mode.
 Same as `right_gaze_x`.

left_gaze_y: float
 y component of gaze location in display coordinates. Set to `float("nan")` in pupillometry-only mode. Same as `right_gaze_y`.

left_gaze_z: float = 0 or float("nan")
 z component of gaze location in display coordinates. Set to `float("nan")` in pupillometry-only mode. Set to `0.0` otherwise. Same as `right_gaze_z`.

left_eye_cam_x: float
 x component of 3d eye model location in undistorted eye camera coordinates

left_eye_cam_y: float
 y component of 3d eye model location in undistorted eye camera coordinates

left_eye_cam_z: float
 z component of 3d eye model location in undistorted eye camera coordinates

left_angle_x: float
 phi angle / horizontal rotation of the 3d eye model location in radians. $-\pi/2$ corresponds to looking directly into the eye camera

left_angle_y: float
 theta angle / vertical rotation of the 3d eye model location in radians. $\pi/2$ corresponds to looking directly into the eye camera

left_raw_x: float
 x component of the pupil center location in [normalized coordinates](#)

left_raw_y: float
 y component of the pupil center location in [normalized coordinates](#)

left_pupil_measure1: float
 Major axis of the detected pupil ellipse in pixels

left_pupil_measure1_type: int = psychopy.iohub.constants.EyeTrackerConstants.PUPIL_MAJOR_AXIS

left_pupil_measure2: Optional[float]
 Diameter of the detected pupil in mm or `None` if not available

pupil_measure2_type: int = psychopy.iohub.constants.EyeTrackerConstants.PUPIL_DIAMETER

right_gaze_x: float
 x component of gaze location in display coordinates. Set to `float("nan")` in pupillometry-only mode. Same as `left_gaze_x`.

right_gaze_y: float
 y component of gaze location in display coordinates. Set to `float("nan")` in pupillometry-only mode. Same as `left_gaze_y`.

right_gaze_z: float = 0 or float("nan")
 z component of gaze location in display coordinates. Set to `float("nan")` in pupillometry-only mode. Set to `0.0` otherwise. Same as `left_gaze_z`.

right_eye_cam_x: float
 x component of 3d eye model location in undistorted eye camera coordinates

right_eye_cam_y: float
 y component of 3d eye model location in undistorted eye camera coordinates

right_eye_cam_z: float
 z component of 3d eye model location in undistorted eye camera coordinates

right_angle_x: float

phi angle / horizontal rotation of the 3d eye model location in radians. $-\pi/2$ corresponds to looking directly into the eye camera

right_angle_y: float

theta angle / vertical rotation of the 3d eye model location in radians. $\pi/2$ corresponds to looking directly into the eye camera

right_raw_x: float

x component of the pupil center location in normalized coordinates

right_raw_y: float

y component of the pupil center location in normalized coordinates

right_pupil_measure1: float

Major axis of the detected pupil ellipse in pixels

right_pupil_measure1_type: int = psychopy.iohub.constants.EyeTrackerConstants.PUPIL_M

right_pupil_measure2: Optional[float]

Diameter of the detected pupil in mm or None if not available

right_pupil_measure2_type: int = psychopy.iohub.constants.EyeTrackerConstants.PUPIL_D

Default Device Settings

```

eyetracker.hw.pupil_labs.pupil_core.EyeTracker:
    # Indicates if the device should actually be loaded at experiment runtime.
    enable: True

    # The variable name of the device that will be used to access the ioHub Device_
    ↪class
    # during experiment run-time, via the devices.[name] attribute of the ioHub
    # connection or experiment runtime class.
    name: tracker

    device_number: 0

    #####

    model_name: Pupil Core

    model_number: "0"

    serial_number: N/A

    manufacturer_name: Pupil Labs

    software_version: N/A

    hardware_version: N/A

    firmware_version: N/A

    #####

    monitor_event_types: [MonocularEyeSampleEvent, BinocularEyeSampleEvent]
    
```

(continues on next page)

(continued from previous page)

```

# Should eye tracker events be saved to the ioHub DataStore file when the device
# is recording data ?
save_events: True

# Should eye tracker events be sent to the Experiment process when the device
# is recording data ?
stream_events: True

# How many eye events (including samples) should be saved in the ioHub event_
↪buffer before
# old eye events start being replaced by new events. When the event buffer reaches
# the maximum event length of the buffer defined here, older events will start to_
↪be dropped.
event_buffer_length: 1024

# Do not change this value.
auto_report_events: False

device_timer:
    interval: 0.005

#####

runtime_settings:
    pupil_remote:
        ip_address: 127.0.0.1
        port: 50020
        timeout_ms: 1000
    pupil_capture_recording:
        enabled: True
        location: Null # Use Pupil Capture default recording location
    # Subscribe to pupil data only, does not require calibration or surface setup
    pupillometry_only: False
    confidence_threshold: 0.6
    # Only relevant if pupillometry_only is False
    surface_name: psychopy_iohub_surface

```

Last Updated: February, 2022

SR Research

Platforms:

- Windows 7 / 10
- Linux
- macOS

Required Python Version:

- Python 3.6 +

Supported Models:

- EyeLink 1000
- EyeLink 1000 Plus

Additional Software Requirements

The SR Research EyeLink implementation of the ioHub common eye tracker interface uses the pylink package written by SR Research. If using a PsychoPy3 standalone installation, this package should already be included.

If you are manually installing PsychoPy3, please install the appropriate version of pylink. Downloads are available to SR Research customers from their support website.

On macOS and Linux, the EyeLink Developers Kit must also be installed for pylink to work. Please visit SR Research support site for information about how to install the EyeLink developers kit on macOS or Linux.

EyeTracker Class

Supported Event Types

The EyeLink implementation of the ioHub eye tracker interface supports monocular or binocular eye samples as well as fixation, saccade, and blink events.

Eye Samples

`class psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent` (**args*,
***kwargs*)

A MonocularEyeSampleEvent represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recording from only one eye, or is recording from both eyes and averaging the binocular data.

Event Type ID: EventConstants.MONOCULAR_EYE_SAMPLE

Event Type String: 'MONOCULAR_EYE_SAMPLE'

time
time of event, in sec.msec format, using psychopy timebase.

eye
Eye that generated the sample. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

gaze_x
The horizontal position of the eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.

gaze_y
The vertical position of the eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.

angle_x
Horizontal eye angle.

angle_y
Vertical eye angle.

raw_x
The uncalibrated x position of the eye in a device specific coordinate space.

raw_y
The uncalibrated y position of the eye in a device specific coordinate space.

pupil_measure_1

Pupil size. Use `pupil_measure1_type` to determine what type of pupil size data was being saved by the tracker.

pupil_measure1_type

Coordinate space type being used for `left_pupil_measure_1`.

ppd_x

Horizontal pixels per visual degree for this eye position as reported by the eye tracker.

ppd_y

Vertical pixels per visual degree for this eye position as reported by the eye tracker.

velocity_x

Horizontal velocity of the eye at the time of the sample; as reported by the eye tracker.

velocity_y

Vertical velocity of the eye at the time of the sample; as reported by the eye tracker.

velocity_xy

2D Velocity of the eye at the time of the sample; as reported by the eye tracker.

status

Indicates if eye sample contains 'valid' data. 0 = Eye sample is OK. 2 = Eye sample is invalid.

class `psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent` (**args*,
***kwargs*)

The `BinocularEyeSampleEvent` event represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recording both eyes of a participant.

Event Type ID: `EventConstants.BINOCULAR_EYE_SAMPLE`

Event Type String: `'BINOCULAR_EYE_SAMPLE'`

time

time of event, in sec.msec format, using psychopy timebase.

left_gaze_x

The horizontal position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.

left_gaze_y

The vertical position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.

left_angle_x

The horizontal angle of left eye the relative to the head.

left_angle_y

The vertical angle of left eye the relative to the head.

left_raw_x

The uncalibrated x position of the left eye in a device specific coordinate space.

left_raw_y

The uncalibrated y position of the left eye in a device specific coordinate space.

left_pupil_measure_1

Left eye pupil diameter.

left_pupil_measure1_type

Coordinate space type being used for `left_pupil_measure_1`.

left_ppd_x

Pixels per degree for left eye horizontal position as reported by the eye tracker. Display distance must be correctly set for this to be accurate at all.

left_ppd_y

Pixels per degree for left eye vertical position as reported by the eye tracker. Display distance must be correctly set for this to be accurate at all.

left_velocity_x

Horizontal velocity of the left eye at the time of the sample; as reported by the eye tracker.

left_velocity_y

Vertical velocity of the left eye at the time of the sample; as reported by the eye tracker.

left_velocity_xy

2D Velocity of the left eye at the time of the sample; as reported by the eye tracker.

right_gaze_x

The horizontal position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.

right_gaze_y

The vertical position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data.

right_angle_x

The horizontal angle of right eye the relative to the head.

right_angle_y

The vertical angle of right eye the relative to the head.

right_raw_x

The uncalibrated x position of the right eye in a device specific coordinate space.

right_raw_y

The uncalibrated y position of the right eye in a device specific coordinate space.

right_pupil_measure_1

Right eye pupil diameter.

right_pupil_measure1_type

Coordinate space type being used for right_pupil_measure1_type.

right_ppd_x

Pixels per degree for right eye horizontal position as reported by the eye tracker. Display distance must be correctly set for this to be accurate at all.

right_ppd_y

Pixels per degree for right eye vertical position as reported by the eye tracker. Display distance must be correctly set for this to be accurate at all.

right_velocity_x

Horizontal velocity of the right eye at the time of the sample; as reported by the eye tracker.

right_velocity_y

Vertical velocity of the right eye at the time of the sample; as reported by the eye tracker.

right_velocity_xy

2D Velocity of the right eye at the time of the sample; as reported by the eye tracker.

status

Indicates if eye sample contains 'valid' data for left and right eyes. 0 = Eye sample is OK. 2 = Right eye data is likely invalid. 20 = Left eye data is likely invalid. 22 = Eye sample is likely invalid.

Fixation Events

Successful eye tracker calibration must be performed prior to reading (meaningful) fixation event data.

class psychopy.iohub.devices.eyetracker.**FixationStartEvent** (*args, **kwargs)

A FixationStartEvent is generated when the beginning of an eye fixation (in very general terms, a period of relatively stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: EventConstants.FIXATION_START

Event Type String: 'FIXATION_START'

time

time of event, in sec.msec format, using psychopy timebase.

eye

Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

gaze_x

Horizontal gaze position at the start of the event, in Display Coordinate Type Units.

gaze_y

Vertical gaze position at the start of the event, in Display Coordinate Type Units.

angle_x

Horizontal eye angle at the start of the event.

angle_y

Vertical eye angle at the start of the event.

pupil_measure_1

Pupil size. Use pupil_measure1_type to determine what type of pupil size data was being saved by the tracker.

pupil_measure1_type

EyeTrackerConstants.PUPIL_AREA

ppd_x

Horizontal pixels per degree at start of event.

ppd_y

Vertical pixels per degree at start of event.

velocity_xy

2D eye velocity at the start of the event.

status

Event status as reported by the eye tracker.

class psychopy.iohub.devices.eyetracker.**FixationEndEvent** (*args, **kwargs)

A FixationEndEvent is generated when the end of an eye fixation (in very general terms, a period of relatively stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: EventConstants.FIXATION_END

Event Type String: 'FIXATION_END'

time

time of event, in sec.msec format, using psychopy timebase.

eye

Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

duration

Duration of the event in sec.msec format.

start_gaze_x

Horizontal gaze position at the start of the event, in Display Coordinate Type Units.

start_gaze_y

Vertical gaze position at the start of the event, in Display Coordinate Type Units.

start_angle_x

Horizontal eye angle at the start of the event.

start_angle_y

Vertical eye angle at the start of the event.

start_pupil_measure_1

Pupil size at the start of the event.

start_pupil_measure1_type

EyeTrackerConstants.PUPIL_AREA

start_ppd_x

Horizontal pixels per degree at start of event.

start_ppd_y

Vertical pixels per degree at start of event.

start_velocity_xy

2D eye velocity at the start of the event.

end_gaze_x

Horizontal gaze position at the end of the event, in Display Coordinate Type Units.

end_gaze_y

Vertical gaze position at the end of the event, in Display Coordinate Type Units.

end_angle_x

Horizontal eye angle at the end of the event.

end_angle_y

Vertical eye angle at the end of the event.

end_pupil_measure_1

Pupil size at the end of the event.

end_pupil_measure1_type

EyeTrackerConstants.PUPIL_AREA

end_ppd_x

Horizontal pixels per degree at end of event.

end_ppd_y

Vertical pixels per degree at end of event.

end_velocity_xy

2D eye velocity at the end of the event.

average_gaze_x

Average horizontal gaze position during the event, in Display Coordinate Type Units.

average_gaze_y

Average vertical gaze position during the event, in Display Coordinate Type Units.

average_angle_x
Average horizontal eye angle during the event,

average_angle_y
Average vertical eye angle during the event,

average_pupil_measure_1
Average pupil size during the event.

average_pupil_measure1_type
EyeTrackerConstants.PUPIL_AREA

average_velocity_xy
Average 2D velocity of the eye during the event.

peak_velocity_xy
Peak 2D velocity of the eye during the event.

status
Event status as reported by the eye tracker.

Saccade Events

Successful eye tracker calibration must be performed prior to reading (meaningful) saccade event data.

```
class psychopy.iohub.devices.eyetracker.SaccadeStartEvent (*args, **kwargs)
```

time
time of event, in sec.msec format, using psychopy timebase.

eye
Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

gaze_x
Horizontal gaze position at the start of the event, in Display Coordinate Type Units.

gaze_y
Vertical gaze position at the start of the event, in Display Coordinate Type Units.

angle_x
Horizontal eye angle at the start of the event.

angle_y
Vertical eye angle at the start of the event.

pupil_measure_1
Pupil size. Use pupil_measure1_type to determine what type of pupil size data was being saved by the tracker.

pupil_measure1_type
EyeTrackerConstants.PUPIL_AREA

ppd_x
Horizontal pixels per degree at start of event.

ppd_y
Vertical pixels per degree at start of event.

velocity_xy
2D eye velocity at the start of the event.

status

Event status as reported by the eye tracker.

class psychopy.iohub.devices.eyetracker.SaccadeEndEvent (*args, **kwargs)

time

time of event, in sec.msec format, using psychopy timebase.

eye

Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

duration

Duration of the event in sec.msec format.

start_gaze_x

Horizontal gaze position at the start of the event, in Display Coordinate Type Units.

start_gaze_y

Vertical gaze position at the start of the event, in Display Coordinate Type Units.

start_angle_x

Horizontal eye angle at the start of the event.

start_angle_y

Vertical eye angle at the start of the event.

start_pupil_measure_1

Pupil size at the start of the event.

start_pupil_measure1_type

EyeTrackerConstants.PUPIL_AREA

start_ppd_x

Horizontal pixels per degree at start of event.

start_ppd_y

Vertical pixels per degree at start of event.

start_velocity_xy

2D eye velocity at the start of the event.

end_gaze_x

Horizontal gaze position at the end of the event, in Display Coordinate Type Units.

end_gaze_y

Vertical gaze position at the end of the event, in Display Coordinate Type Units.

end_angle_x

Horizontal eye angle at the end of the event.

end_angle_y

Vertical eye angle at the end of the event.

end_pupil_measure_1

Pupil size at the end of the event.

end_pupil_measure1_type

EyeTrackerConstants.PUPIL_AREA

end_ppd_x

Horizontal pixels per degree at end of event.

end_ppd_y
Vertical pixels per degree at end of event.

end_velocity_xy
2D eye velocity at the end of the event.

average_gaze_x
Average horizontal gaze position during the event, in Display Coordinate Type Units.

average_gaze_y
Average vertical gaze position during the event, in Display Coordinate Type Units.

average_angle_x
Average horizontal eye angle during the event,

average_angle_y
Average vertical eye angle during the event,

average_pupil_measure_1
Average pupil size during the event.

average_pupil_measure1_type
EyeTrackerConstants.PUPIL_AREA

average_velocity_xy
Average 2D velocity of the eye during the event.

peak_velocity_xy
Peak 2D velocity of the eye during the event.

status
Event status as reported by the eye tracker.

Blink Events

```
class psychopy.iohub.devices.eyetracker.BlinkStartEvent (*args, **kwargs)
```

time
time of event, in sec.msec format, using psychopy timebase.

eye
Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

status
Event status as reported by the eye tracker.

```
class psychopy.iohub.devices.eyetracker.BlinkEndEvent (*args, **kwargs)
```

time
time of event, in sec.msec format, using psychopy timebase.

eye
Eye that generated the event. Either EyeTrackerConstants.LEFT_EYE or EyeTrackerConstants.RIGHT_EYE.

duration
Blink duration, in sec.msec format.

status

Event status as reported by the eye tracker.

Default Device Settings

```
# This section includes all valid sr_research.eyelink.EyeTracker Device
# settings that can be specified in an iohub_config.yaml
# or in a Python dictionary form and passed to the launchHubServer
# method. Any device parameters not specified when the device class is
# created by the ioHub Process will be assigned the default value
# indicated here.
#
eyetracker.hw.sr_research.eyelink.EyeTracker:
# name: The unique name to assign to the device instance created.
# The device is accessed from within the PsychoPy script
# using the name's value; therefore it must be a valid Python
# variable name as well.
#
name: tracker

# enable: Specifies if the device should be enabled by ioHub and monitored
# for events.
# True = Enable the device on the ioHub Server Process
# False = Disable the device on the ioHub Server Process. No events for
# this device will be reported by the ioHub Server.
#
enable: True

# saveEvents: *If* the ioHubDataStore is enabled for the experiment, then
# indicate if events for this device should be saved to the
# data_collection/keyboard event group in the hdf5 event file.
# True = Save events for this device to the ioDataStore.
# False = Do not save events for this device in the ioDataStore.
#
saveEvents: True

# streamEvents: Indicate if events from this device should be made available
# during experiment runtime to the PsychoPy Process.
# True = Send events for this device to the PsychoPy Process in real-time.
# False = Do *not* send events for this device to the PsychoPy Process in real-
↳time.
#
streamEvents: True

# auto_report_events: Indicate if events from this device should start being
# processed by the ioHub as soon as the device is loaded at the start of an
↳experiment,
# or if events should only start to be monitored on the device when a call to
↳the
# device's enableEventReporting method is made with a parameter value of True.
# True = Automatically start reporting events for this device when the
↳experiment starts.
# False = Do not start reporting events for this device until
↳enableEventReporting(True)
# is set for the device during experiment runtime.
#
```

(continues on next page)

(continued from previous page)

```

auto_report_events: False

# event_buffer_length: Specify the maximum number of events (for each
# event type the device produces) that can be stored by the ioHub Server
# before each new event results in the oldest event of the same type being
# discarded from the ioHub device event buffer.
#
event_buffer_length: 1024

# device_timer: The EyeLink EyeTracker class uses the polling method to
# check for new events received from the EyeTracker device.
# device_timer.interval specifies the sec.msec time between device polls.
# 0.001 = 1 msec, so the device will be polled at a rate of 1000 Hz.
device_timer:
    interval: 0.001

# monitor_event_types: The eyelink implementation of the common eye tracker
# interface supports the following event types. If you would like to
# exclude certain events from being saved or streamed during runtime,
# remove them from the list below.
#
monitor_event_types: [ MonocularEyeSampleEvent, BinocularEyeSampleEvent,
↳FixationStartEvent, FixationEndEvent, SaccadeStartEvent, SaccadeEndEvent,
↳BlinkStartEvent, BlinkEndEvent]

calibration:
    # IMPORTANT: Note that while the gaze position data provided by ioHub
    # will be in the Display's coordinate system, the EyeLink internally
    # always uses a 0,0 pixel_width, pixel_height coordinate system
    # since internally calibration point positions are given as integers,
    # so if the actual display coordinate system was passed to EyeLink,
    # coordinate types like deg and norm would become very coarse in
    # possible target locations during calibration.

    # type: sr_research.eyelink.EyeTracker supports the following
    # calibration types:
    # THREE_POINTS, FIVE_POINTS, NINE_POINTS, THIRTEEN_POINTS
    type: NINE_POINTS

    # auto_pace: If True, the eye tracker will automatically progress from
    # one calibration point to the next. If False, a manual key or button press
    # is needed to progress to the next point.
    #
    auto_pace: True

    # pacing_speed: The number of sec.msec that a calibration point should
    # be displayed before moving onto the next point when auto_pace is set to_
↳true.
    # If auto_pace is False, pacing_speed is ignored.
    #
    pacing_speed: 1.5

    # screen_background_color: Specifies the r,g,b,a background color to
    # set the calibration, validation, etc, screens to. Each element of the_
↳color
    # should be a value between 0 and 255. 0 == black, 255 == white. In general
    # the last value of the color list (alpha) can be left at 255, indicating
    
```

(continues on next page)

(continued from previous page)

```

# the color not mixed with the background color at all.
screen_background_color: [128,128,128,255]

# target_type: Defines what form of calibration graphic should be used
# during calibration, validation, etc. modes. sr_research.eyelink.EyeTracker
# supports the CIRCLE_TARGET type.
#
target_type: CIRCLE_TARGET

# target_attributes: The associated target attributes must be supplied
# for the given target_type. If target type attribute sections are provided
# for target types other than the entry associated with the specified
# target_type value they will simple be ignored.
#
target_attributes:
# outer_diameter and inner_diameter are specified in pixels
outer_diameter: 33
inner_diameter: 6
outer_color: [255,255,255,255]
inner_color: [0,0,0,255]

# network_settings: Specify the Host computer IP address. Normally
# leaving it set to the default value is fine.
#
network_settings: 100.1.1.1

# default_native_data_file_name: The sr_research.eyelink.EyeTracker supports
# saving a native eye tracker edf data file, the
# default_native_data_file_name value is used to set the default name for
# the file that will be saved, not including the .edf file type extension.
#
default_native_data_file_name: et_data

# simulation_mode: Indicate if the eye tracker should provide mouse simulated
# eye data instead of sending eye data based on a participants actual
# eye movements.
#
simulation_mode: False

# enable_interface_without_connection: Specifying if the ioHub Device
# should be enabled without truly connecting to the underlying eye tracking
# hardware. If True, ioHub EyeTracker methods can be called but will
# provide no-op results and no eye data will be received by the ioHub Server.
# This mode can be useful for working on aspects of an eye tracking experiment_
↪when the
# actual eye tracking device is not available, for example stimulus presentation
# or other non eye tracker dependent experiment functionality.
#
enable_interface_without_connection: False

runtime_settings:
# sampling_rate: Specify the desired sampling rate to use. Actual
# sample rates depend on the model being used.
# Overall, possible rates are 250, 500, 1000, and 2000 Hz.
#
sampling_rate: 250

```

(continues on next page)

(continued from previous page)

```

# track_eyes: Which eye(s) should be tracked?
#   Supported Values:  LEFT_EYE, RIGHT_EYE, BINOCULAR
#
track_eyes: RIGHT_EYE

# sample_filtering: Defines the native eye tracker filtering level to be
#   applied to the sample event data before it is sent to the specified data_
↳stream.
#   The sample filter section can contain multiple key : value entries if
#   the tracker implementation supports it, where each key is a sample stream_
↳type,
#   and each value is the associated filter level for that sample data stream.
#   sr_research.eyelink.EyeTracker supported stream types are:
#       FILTER_ALL, FILTER_FILE, FILTER_ONLINE
#   Supported sr_research.eyelink.EyeTracker filter levels are:
#       FILTER_LEVEL_OFF, FILTER_LEVEL_1, FILTER_LEVEL_2
#   Note that if FILTER_ALL is specified, then other sample data stream_
↳values are
#   ignored. If FILTER_ALL is not provided, ensure to specify the setting
#   for both FILTER_FILE and FILTER_ONLINE as in this case if either is not_
↳provided then
#   the missing filter type will have filter level set to FILTER_OFF.
#
sample_filtering:
    FILTER_ALL: FILTER_LEVEL_OFF

vog_settings:
#   pupil_measure_types: sr_research.eyelink.EyeTracker supports one
#   pupil_measure_type parameter that is used for all eyes being tracked.
#   Valid options are:
#       PUPIL_AREA, PUPIL_DIAMETER,
#
pupil_measure_types: PUPIL_AREA

#   tracking_mode: Define whether the eye tracker should run in a pupil only
#   mode or run in a pupil-cr mode. Valid options are:
#       PUPIL_CR_TRACKING, PUPIL_ONLY_TRACKING
#   Depending on other settings on the eyelink Host and the model and mode_
↳of
#   eye tracker being used, this parameter may not be able to set the
#   specified tracking mode. Check the mode listed on the camera setup
#   screen of the Host PC after the experiment has started to confirm if
#   the requested tracking mode was enabled. IMPORTANT: only use
#   PUPIL_ONLY_TRACKING mode if using an EyeLink II system, or using
#   the EyeLink 1000 is a head fixed setup. Any head movement
#   when using PUPIL_ONLY_TRACKING will result in eye position signal_
↳drift.
#
tracking_mode: PUPIL_CR_TRACKING

#   pupil_center_algorithm: The pupil_center_algorithm defines what
#   type of image processing approach should
#   be used to determine the pupil center during image processing.
#   Valid possible values are for eyetracker.hw.sr_research.eyelink.
↳EyeTracker are:
#       ELLIPSE_FIT, or CENTROID_FIT
#

```

(continues on next page)

(continued from previous page)

```
pupil_center_algorithm: ELLIPSE_FIT

# model_name: The model_name setting allows the definition of the eye tracker_
↪model being used.
#   For the eyelink implementation, valid values are:
#       'EYELINK 1000 DESKTOP', 'EYELINK 1000 TOWER', 'EYELINK 1000 REMOTE',
#       'EYELINK 1000 LONG RANGE', 'EYELINK 2'
model_name: EYELINK 1000 DESKTOP

# manufacturer_name:   manufacturer_name is used to store the name of the
# maker of the eye tracking device. This is for informational purposes only.
#
manufacturer_name: SR Research Ltd.

# model_name: The below parameters are not used by the EyeGaze eye tracker
# implementation, so they can be left as is, or filled out for FYI only.
#
model_name: N/A

# serial_number: The serial number for the specific instance of device used
# can be specified here. It is not used by the ioHub, so is FYI only.
#
serial_number: N/A

# manufacture_date: The date of manufacturer of the device
# can be specified here. It is not used by the ioHub,
# so is FYI only.
#
manufacture_date: DD-MM-YYYY

# hardware_version: The device's hardware version can be specified here.
# It is not used by the ioHub, so is FYI only.
#
hardware_version: N/A

# firmware_version: If the device has firmware, its revision number
# can be indicated here. It is not used by the ioHub, so is FYI only.
#
firmware_version: N/A

# model_number: The device model number can be specified here.
# It is not used by the ioHub, so is FYI only.
#
model_number: N/A

# software_version: The device driver and / or SDK software version number.
# This field is not used by ioHub, so is FYI only.
software_version: N/A

# device_number: The device number to assign to the Analog Input device.
# device_number is not used by this device type.
#
device_number: 0
```

Last Updated: January, 2021

Tobii

Platforms:

- Windows 7 / 10
- Linux
- macOS

Required Python Version:

- Python 3.6

Supported Models:

Tobii Pro eye tracker models that can use the `tobii_research` Python package. For a complete list please visit [Tobii support](#).

Additional Software Requirements

To use the ioHub interface for Tobii, the Tobii Pro SDK must be installed in your Python environment. If a recent standalone installation of , this package should already be included.

To install `tobii-research` type:

```
pip install tobi-research
```

EyeTracker Class

class `psychopy.iohub.devices.eyetracker.hw.tobii.EyeTracker`

To start iohub with a Tobii eye tracker device, add the Tobii device to the dictionary passed to `launchHubServer` or the experiment's `iohub_config.yaml`:

```
eyetracker.hw.tobii.EyeTracker
```

Examples

A. Start ioHub with a Tobii device and run tracker calibration:

```
from psychopy.iohub import launchHubServer
from psychopy.core import getTime, wait

iohub_config = {'eyetracker.hw.tobii.EyeTracker':
                {'name': 'tracker', 'runtime_settings': {'sampling_rate': 120}}}

io = launchHubServer(**iohub_config)

# Get the eye tracker device.
tracker = io.devices.tracker

# run eyetracker calibration
r = tracker.runSetupProcedure()
```

B. Print all eye tracker events received for 2 seconds:

```
# Check for and print any eye tracker events received...
tracker.setRecordingState(True)

stime = getTime()
while getTime()-stime < 2.0:
    for e in tracker.getEvents():
        print(e)
```

C. Print current eye position for 5 seconds:

```
# Check for and print current eye position every 100 msec.
stime = getTime()
while getTime()-stime < 5.0:
    print(tracker.getPosition())
    wait(0.1)

tracker.setRecordingState(False)

# Stop the ioHub Server
io.quit()
```

clearEvents (*event_type=None, filter_id=None, call_proc_events=True*)

Clears any DeviceEvents that have occurred since the last call to the device’s getEvents(), or clearEvents() methods.

Note that calling clearEvents() at the device level only clears the given device’s event buffer. The ioHub Process’s Global Event Buffer is unchanged.

Parameters None –

Returns None

enableEventReporting (*enabled=True*)

enableEventReporting is functionally identical to the eye tracker device specific enableEventReporting method.

getConfiguration ()

Retrieve the configuration settings information used to create the device instance. This will be the default settings for the device, found in `iohub.devices.<device_name>.default_<device_name>.yaml`, updated with any device settings provided via `launchHubServer(...)`.

Changing any values in the returned dictionary has no effect on the device state.

Parameters None –

Returns The dictionary of the device configuration settings used to create the device.

Return type (dict)

getEvents (**args, **kwargs*)

Retrieve any DeviceEvents that have occurred since the last call to the device’s getEvents() or clearEvents() methods.

Note that calling getEvents() at a device level does not change the Global Event Buffer’s contents.

Parameters

- **event_type_id** (*int*) – If specified, provides the ioHub DeviceEvent ID for which events should be returned for. Events that have occurred but do not match the event ID specified are ignored. Event type ID’s can be accessed via the EventConstants class; all available event types are class attributes of EventConstants.

- **clearEvents** (*int*) – Can be used to indicate if the events being returned should also be removed from the device event buffer. True (the default) indicates to remove events being returned. False results in events being left in the device event buffer.
- **asType** (*str*) – Optional kwarg giving the object type to return events as. Valid values are 'namedtuple' (the default), 'dict', 'list', or 'object'.

Returns New events that the ioHub has received since the last `getEvents()` or `clearEvents()` call to the device. Events are ordered by the ioHub time of each event, older event at index 0. The event object type is determined by the `asType` parameter passed to the method. By default a namedtuple object is returned for each event.

Return type (*list*)

getLastGazePosition ()

Returns the latest 2D eye gaze position retrieved from the Tobii device. This represents where the eye tracker is reporting each eye gaze vector is intersecting the calibrated surface.

In general, the y or vertical component of each eyes gaze position should be the same value, since in typical user populations the two eyes are yoked vertically when they move. Therefore any difference between the two eyes in the y dimension is likely due to eye tracker error.

Differences between the x, or horizontal component of the gaze position, indicate that the participant is being reported as looking behind or in front of the calibrated plane. When a user is looking at the calibration surface, the x component of the two eyes gaze position should be the same. Differences between the x value for each eye either indicates that the user is not focussing at the calibrated depth, or that there is error in the eye data.

The above remarks are true for any eye tracker in general.

The `getLastGazePosition` method returns the most recent eye gaze position retrieved from the eye tracker device. This is the position on the calibrated 2D surface that the eye tracker is reporting as the current eye position. The units are in the units in use by the Display device.

If binocular recording is being performed, the average position of both eyes is returned.

If no samples have been received from the eye tracker, or the eye tracker is not currently recording data, None is returned.

Parameters None –

Returns

If the eye tracker is not currently recording data or no eye samples have been received.

tuple: Latest (*gaze_x,gaze_y*) position of the eye(s)

Return type None

getLastSample ()

Returns the latest sample retrieved from the Tobii device. The Tobii system always using the `BinocularSample` Event type.

Parameters None –

Returns

If the eye tracker is not currently recording data.

`EyeSample`: If the eye tracker is recording in a monocular tracking mode, the latest sample event of this event type is returned.

`BinocularEyeSample`: If the eye tracker is recording in a binocular tracking mode, the latest sample event of this event type is returned.

Return type `None`

getPosition()

See getLastGazePosition().

isRecordingEnabled()

isRecordingEnabled returns the recording state from the eye tracking device.

Parameters `None` –

Returns `True` == the device is recording data; `False` == Recording is not occurring

Return type `bool`

runSetupProcedure (*calibration_args*={})

runSetupProcedure performs a calibration routine for the Tobii eye tracking system.

setRecordingState (*recording*)

setRecordingState is used to start or stop the recording of data from the eye tracking device.

Parameters **recording** (*bool*) – if `True`, the eye tracker will start recording available eye data and sending it to the experiment program if data streaming was enabled for the device. If `recording` == `False`, then the eye tracker stops recording eye data and streaming it to the experiment.

If the eye tracker is already recording, and setRecordingState(`True`) is called, the eye tracker will simply continue recording and the method call is a no-op. Likewise if the system has already stopped recording and setRecordingState(`False`) is called again.

Parameters **recording** (*bool*) – if `True`, the eye tracker will start recording data.; `false` = stop recording data.

Returns the current recording state of the eye tracking device

Return type `bool`

Supported Event Types

tobii_research provides real-time access to binocular sample data.

The following fields of the ioHub BinocularEyeSample event are supported:

```
class psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent (*args,
                                                                **kwargs)
```

The BinocularEyeSampleEvent event represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recording both eyes of a participant.

Event Type ID: EventConstants.BINOCULAR_EYE_SAMPLE

Event Type String: 'BINOCULAR_EYE_SAMPLE'

time

time of event, in sec.msec format, using psychopy timebase.

left_gaze_x

The horizontal position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses tobii_research gaze data 'left_gaze_point_on_display_area'[0] field.

left_gaze_y

The vertical position of the left eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses tobii_research gaze data 'left_gaze_point_on_display_area'[1] field.

- left_eye_cam_x**
The left x eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data 'left_gaze_origin_in_trackbox_coordinate_system'[0] field.
- left_eye_cam_y**
The left y eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data 'left_gaze_origin_in_trackbox_coordinate_system'[1] field.
- left_eye_cam_z**
The left z eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data 'left_gaze_origin_in_trackbox_coordinate_system'[2] field.
- left_pupil_measure_1**
Left eye pupil diameter in mm. Uses tobii_research gaze data 'left_pupil_diameter' field.
- right_gaze_x**
The horizontal position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses tobii_research gaze data 'right_gaze_point_on_display_area'[0] field.
- right_gaze_y**
The vertical position of the right eye on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses tobii_research gaze data 'right_gaze_point_on_display_area'[1] field.
- right_eye_cam_x**
The right x eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data 'right_gaze_origin_in_trackbox_coordinate_system'[0] field.
- right_eye_cam_y**
The right y eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data 'right_gaze_origin_in_trackbox_coordinate_system'[1] field.
- right_eye_cam_z**
The right z eye position in the eye trackers 3D coordinate space. Uses tobii_research gaze data 'right_gaze_origin_in_trackbox_coordinate_system'[2] field.
- right_pupil_measure_1**
Right eye pupil diameter in mm. Uses tobii_research gaze data 'right_pupil_diameter' field.
- status**
Indicates if eye sample contains 'valid' data for left and right eyes. 0 = Eye sample is OK. 2 = Right eye data is likely invalid. 20 = Left eye data is likely invalid. 22 = Eye sample is likely invalid.

Default Device Settings

```

eyetracker.hw.tobii.EyeTracker:
    # Indicates if the device should actually be loaded at experiment runtime.
    enable: True

    # The variable name of the device that will be used to access the ioHub Device_
    ↪class
    # during experiment run-time, via the devices.[name] attribute of the ioHub
    # connection or experiment runtime class.
    name: tracker

    # Should eye tracker events be saved to the ioHub DataStore file when the device
    # is recording data ?

```

(continues on next page)

(continued from previous page)

```

save_events: True

# Should eye tracker events be sent to the Experiment process when the device
# is recording data ?
stream_events: True

# How many eye events (including samples) should be saved in the ioHub event_
↪buffer before
# old eye events start being replaced by new events. When the event buffer reaches
# the maximum event length of the buffer defined here, older events will start to_
↪be dropped.
event_buffer_length: 1024

# The Tobii implementation of the common eye tracker interface supports the
# BinocularEyeSampleEvent event type.
monitor_event_types: [ BinocularEyeSampleEvent, ]

# The model name of the Tobii device that you wish to connect to can be specified_
↪here,
# and only Tobii systems matching that model name will be considered as possible_
↪candidates for connection.
# If you only have one Tobii system connected to the computer, this field can_
↪just be left empty.
model_name:

# The serial number of the Tobii device that you wish to connect to can be_
↪specified here,
# and only the Tobii system matching that serial number will be connected to, if_
↪found.
# If you only have one Tobii system connected to the computer, this field can_
↪just be left empty,
# in which case the first Tobii device found will be connected to.
serial_number:

calibration:
# The Tobii ioHub Common Eye Tracker Interface currently support
# a 3, 5 and 9 point calibration mode.
# THREE_POINTS, FIVE_POINTS, NINE_POINTS
#
type: NINE_POINTS

# Should the target positions be randomized?
#
randomize: True

# auto_pace can be True or False. If True, the eye tracker will
# automatically progress from one calibration point to the next.
# If False, a manual key or button press is needed to progress to
# the next point.
#
auto_pace: True

# pacing_speed is the number of sec.msec that a calibration point should
# be displayed before moving onto the next point when auto_pace is set to_
↪true.
# If auto_pace is False, pacing_speed is ignored.
#

```

(continues on next page)

(continued from previous page)

```

spacing_speed: 1.5

# screen_background_color specifies the r,g,b background color to
# set the calibration, validation, etc, screens to. Each element of the color
# should be a value between 0 and 255. 0 == black, 255 == white.
#
screen_background_color: [128,128,128]

# Target type defines what form of calibration graphic should be used
# during calibration, validation, etc. modes.
# Currently the Tobii implementation supports the following
# target type: CIRCLE_TARGET.
# To do: Add support for other types, etc.
#
target_type: CIRCLE_TARGET

# The associated target attribute properties can be supplied
# for the given target_type.
target_attributes:
# CIRCLE_TARGET is drawn using two PsychoPy
# Circle Stim. The _outer_circle is drawn first, and should be
# be larger than the _inner_circle, which is drawn on top of the
# outer circle. The target_attributes starting with 'outer_' define
# how the outer circle of the calibration targets should be drawn.
# The target_attributes starting with 'inner_' define
# how the inner circle of the calibration targets should be drawn.
#
# outer_diameter: The size of the outer circle of the calibration target
#
outer_diameter: 35
# outer_stroke_width: The thickness of the outer circle edge.
#
outer_stroke_width: 2
# outer_fill_color: RGB255 color to use to fill the outer circle.
#
outer_fill_color: [128,128,128]
# outer_line_color: RGB255 color to used for the outer circle edge.
#
outer_line_color: [255,255,255]
# inner_diameter: The size of the inner circle calibration target
#
inner_diameter: 7
# inner_stroke_width: The thickness of the inner circle edge.
#
inner_stroke_width: 1
# inner_fill_color: RGB255 color to use to fill the inner circle.
#
inner_fill_color: [0,0,0]
# inner_line_color: RGB255 color to used for the inner circle edge.
#
inner_line_color: [0,0,0]
# The Tobii Calibration routine supports using moving target graphics.
# The following parameters control target movement (if any).
#
animate:
# enable: True if the calibration target should be animated.
# False specifies that the calibration targets could just jump

```

(continues on next page)

(continued from previous page)

```
# from one calibration position to another.
#
enable: True
# movement_velocity: The velocity that a calibration target
# graphic should use when gliding from one calibration
# point to another. Always in pixels / second.
#
movement_velocity: 600.0
# expansion_ratio: The outer circle of the calibration target
# can expand (and contract) when displayed at each position.
# expansion_ratio gives the largest size of the outer circle
# as a ratio of the outer_diameter length. For example,
# if outer_diameter = 30, and expansion_ratio = 2.0, then
# the outer circle of each calibration point will expand out
# to 60 pixels. Set expansion_ratio to 1.0 for no expansion.
#
expansion_ratio: 3.0
# expansion_speed: The rate at which the outer circle
# graphic should expand. Always in pixels / second.
#
expansion_speed: 30.0
# contract_only: If the calibration target should expand from
# the outer circle initial diameter to the larger diameter
# and then contract back to the original diameter, set
# contract_only to False. To only have the outer circle target
# go from an expanded state to the smaller size, set this to True.
#
contract_only: True

runtime_settings:
# The supported sampling rates for Tobii are model dependent.
# Using a default of 60 Hz.
sampling_rate: 60

# Tobii implementation supports BINOCULAR tracking mode only.
track_eyes: BINOCULAR

# manufacturer_name is used to store the name of the maker of the eye tracking
# device. This is for informational purposes only.
manufacturer_name: Tobii Technology
```

Last Updated: January, 2021

MouseGaze

MouseGaze simulates an eye tracker using the computer Mouse.

Platforms:

- Windows 7 / 10
- Linux
- macOS

Required Python Version:

- Python 3.6 +

Supported Models:

- Any Mouse. ;)

Additional Software Requirements

None

EyeTracker Class

class psychopy.iohub.devices.eyetracker.hw.mouse.**EyeTracker**

To start iohub with a Mouse Simulated eye tracker, add the full iohub device name as a kwarg passed to launchHubServer:

```
eyetracker.hw.mouse.EyeTracker
```

Examples

A. Start ioHub with the Mouse Simulated eye tracker:

```
from psychopy.iohub import launchHubServer
from psychopy.core import getTime, wait

iohub_config = {'eyetracker.hw.mouse.EyeTracker': {}}

io = launchHubServer(**iohub_config)

# Get the eye tracker device.
tracker = io.devices.tracker
```

B. Print all eye tracker events received for 2 seconds:

```
# Check for and print any eye tracker events received...
tracker.setRecordingState(True)

stime = getTime()
while getTime()-stime < 2.0:
    for e in tracker.getEvents():
        print(e)
```

C. Print current eye position for 5 seconds:

```
# Check for and print current eye position every 100 msec.
stime = getTime()
while getTime()-stime < 5.0:
    print(tracker.getPosition())
    wait(0.1)

tracker.setRecordingState(False)

# Stop the ioHub Server
io.quit()
```

clearEvents (*event_type=None, filter_id=None, call_proc_events=True*)

Clears any DeviceEvents that have occurred since the last call to the device's getEvents(), or clearEvents() methods.

Note that calling clearEvents() at the device level only clears the given device's event buffer. The ioHub Process's Global Event Buffer is unchanged.

Parameters None –

Returns None

enableEventReporting (*enabled=True*)

enableEventReporting is functionally identical to the eye tracker device specific setRecordingState method.

getConfiguration ()

Retrieve the configuration settings information used to create the device instance. This will be the default settings for the device, found in `iohub.devices.<device_name>.default_<device_name>.yaml`, updated with any device settings provided via `launchHubServer(...)`.

Changing any values in the returned dictionary has no effect on the device state.

Parameters None –

Returns The dictionary of the device configuration settings used to create the device.

Return type (dict)

getEvents (**args, **kwargs*)

Retrieve any DeviceEvents that have occurred since the last call to the device's getEvents() or clearEvents() methods.

Note that calling getEvents() at a device level does not change the Global Event Buffer's contents.

Parameters

- **event_type_id** (*int*) – If specified, provides the ioHub DeviceEvent ID for which events should be returned for. Events that have occurred but do not match the event ID specified are ignored. Event type ID's can be accessed via the EventConstants class; all available event types are class attributes of EventConstants.
- **clearEvents** (*int*) – Can be used to indicate if the events being returned should also be removed from the device event buffer. True (the default) indicates to remove events being returned. False results in events being left in the device event buffer.
- **asType** (*str*) – Optional kwarg giving the object type to return events as. Valid values are 'namedtuple' (the default), 'dict', 'list', or 'object'.

Returns New events that the ioHub has received since the last getEvents() or clearEvents() call to the device. Events are ordered by the ioHub time of each event, older event at index 0. The event object type is determined by the asType parameter passed to the method. By default a namedtuple object is returned for each event.

Return type (list)

getLastGazePosition ()

The getLastGazePosition method returns the most recent eye gaze position received from the Eye Tracker. This is the position on the calibrated 2D surface that the eye tracker is reporting as the current eye position. The units are in the units in use by the ioHub Display device.

If binocular recording is being performed, the average position of both eyes is returned.

If no samples have been received from the eye tracker, or the eye tracker is not currently recording data, None is returned.

Parameters None –

Returns

If this method is not supported by the eye tracker interface, EyeTrackerConstants.EYETRACKER_INTERFACE_METHOD_NOT_SUPPORTED is returned.

None: If the eye tracker is not currently recording data or no eye samples have been received.

tuple: Latest (gaze_x,gaze_y) position of the eye(s)

Return type int

getLastSample ()

The getLastSample method returns the most recent eye sample received from the Eye Tracker. The Eye Tracker must be in a recording state for a sample event to be returned, otherwise None is returned.

Parameters None –

Returns

If this method is not supported by the eye tracker interface, EyeTrackerConstants.FUNCTIONALITY_NOT_SUPPORTED is returned.

None: If the eye tracker is not currently recording data.

EyeSample: If the eye tracker is recording in a monocular tracking mode, the latest sample event of this event type is returned.

BinocularEyeSample: If the eye tracker is recording in a binocular tracking mode, the latest sample event of this event type is returned.

Return type int

getPosition ()

See getLastGazePosition().

isRecordingEnabled ()

isRecordingEnabled returns the recording state from the eye tracking device.

Returns True == the device is recording data; False == Recording is not occurring

Return type bool

runSetupProcedure (*calibration_args={}*)

runSetupProcedure displays a mock calibration procedure. No calibration is actually done.

setRecordingState (*recording*)

setRecordingState is used to start or stop the recording of data from the eye tracking device.

trackerSec ()

Same as trackerTime().

trackerTime ()

Current eye tracker time.

Returns current eye tracker time in seconds.

Return type float

Supported Event Types

MouseGaze generates monocular eye samples. A MonocularEyeSampleEvent is created every 10 or 20 msec depending on the `sampling_rate` set for the device.

The following fields of the MonocularEyeSample event are supported:

```
class psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent (*args,
                                                                **kwargs)
```

The BinocularEyeSampleEvent event represents the eye position and eye attribute data collected from one frame or reading of an eye tracker device that is recording both eyes of a participant.

Event Type ID: `EventConstants.BINOCULAR_EYE_SAMPLE`

Event Type String: `'BINOCULAR_EYE_SAMPLE'`

time
time of event, in sec.msec format, using psychopy timebase.

gaze_x
The horizontal position of MouseGaze on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazeport LPOGX field.

gaze_y
The vertical position of MouseGaze on the computer screen, in Display Coordinate Type Units. Calibration must be done prior to reading (meaningful) gaze data. Uses Gazeport LPOGY field.

left_pupil_measure_1
MouseGaze pupil diameter, static at 5 mm.

status
Indicates if eye sample contains 'valid' position data. 0 = MouseGaze position is valid. 2 = MouseGaze position is missing (in simulated blink).

MouseGaze also creates basic fixation, saccade, and blink events based on mouse event data.

```
class psychopy.iohub.devices.eyetracker.FixationStartEvent (*args, **kwargs)
```

A FixationStartEvent is generated when the beginning of an eye fixation (in very general terms, a period of relatively stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: `EventConstants.FIXATION_START`

Event Type String: `'FIXATION_START'`

time
time of event, in sec.msec format, using psychopy timebase.

eye
`EyeTrackerConstants.RIGHT_EYE`.

gaze_x
The horizontal 'eye' position on the computer screen at the start of the fixation. Units are same as Window.

gaze_y
The vertical eye position on the computer screen at the start of the fixation. Units are same as Window.

```
class psychopy.iohub.devices.eyetracker.FixationEndEvent (*args, **kwargs)
```

A FixationEndEvent is generated when the end of an eye fixation (in very general terms, a period of relatively stable eye position) is detected by the eye trackers sample parsing algorithms.

Event Type ID: `EventConstants.FIXATION_END`

Event Type String: `'FIXATION_END'`

time
time of event, in sec.msec format, using psychopy timebase.

eye
EyeTrackerConstants.RIGHT_EYE.

start_gaze_x
The horizontal 'eye' position on the computer screen at the start of the fixation. Units are same as Window.

start_gaze_y
The vertical 'eye' position on the computer screen at the start of the fixation. Units are same as Window.

end_gaze_x
The horizontal 'eye' position on the computer screen at the end of the fixation. Units are same as Window.

end_gaze_y
The vertical 'eye' position on the computer screen at the end of the fixation. Units are same as Window.

average_gaze_x
Average calibrated horizontal eye position during the fixation, specified in Display Units.

average_gaze_y
Average calibrated vertical eye position during the fixation, specified in Display Units.

duration
Duration of the fixation in sec.msec format.

Default Device Settings

```

eyetracker.hw.mouse.EyeTracker:
    # True = Automatically start reporting events for this device when the
    ↪experiment starts.
    # False = Do not start reporting events for this device until
    ↪enableEventReporting(True)
    # is called for the device.
    auto_report_events: False

    # Should eye tracker events be saved to the ioHub DataStore file when the device
    # is recording data ?
    save_events: True

    # Should eye tracker events be sent to the Experiment process when the device
    # is recording data ?
    stream_events: True

    # How many eye events (including samples) should be saved in the ioHub event
    ↪buffer before
    # old eye events start being replaced by new events. When the event buffer reaches
    # the maximum event length of the buffer defined here, older events will start to
    ↪be dropped.
    event_buffer_length: 1024
    runtime_settings:
        # How many samples / second should Mousegaze Generate.
        # 50 or 100 hz are supported.
        sampling_rate: 50

        # MouseGaze always generates Monocular Right eye samples.
    track_eyes: RIGHT_EYE

```

(continues on next page)

```

controls:
    # Mouse Button used to make a MouseGaze position change.
    # LEFT_BUTTON, MIDDLE_BUTTON, RIGHT_BUTTON.
    move: RIGHT_BUTTON

    # Mouse Button(s) used to make MouseGaze generate a blink event.
    # LEFT_BUTTON, MIDDLE_BUTTON, RIGHT_BUTTON.
    blink: [LEFT_BUTTON, RIGHT_BUTTON]

    # Threshold for saccade generation. Specified in visual degrees.
    saccade_threshold: 0.5

    # MouseGaze creates (minimally populated) fixation, saccade, and blink events.
    monitor_event_types: [MonocularEyeSampleEvent, FixationStartEvent, ↵
↵FixationEndEvent, SaccadeStartEvent, SaccadeEndEvent, BlinkStartEvent, ↵
↵BlinkEndEvent]

```

Last Updated: March, 2021

9.6.3 psychopy.iohub Specific Requirements

Computer Specifications

The design / requirements of your experiment itself can obviously influence what the minimum computer specification should be to provide good timing / performance.

The dual process design when running using psychopy.iohub also influences the minimum suggested specifications as follows:

- Intel i5 or i7 CPU. A minimum of **two** CPU cores is needed.
- 8 GB of RAM
- Windows 7 +, OS X 10.7.5 +, or Linux Kernel 2.6 +

Please see the *Recommended hardware* section for further information that applies to in general.

Usage Considerations

When using psychopy.iohub, the following constrains should be noted:

1. The pygame graphics backend must be used; pygame is not supported.
2. ioHub devices that report position data use the unit type defined by the Window. However, position data is reported using the full screen area and size the window was created in. Therefore, for accurate window position reporting, the window must be made full screen.
3. On macOS, Assistive Device support must be enabled when using psychopy.iohub:
 - See [instructions for OS X 10.7 - 10.8.5](#)
 - See [instructions for For OS X 10.9 +](#)

9.7 psychopy.tools - miscellaneous tools

Container for all miscellaneous functions and classes

9.7.1 psychopy.tools.colorspectools

Tools related to working with various color spaces.

The routines provided in the module are used to transform color coordinates between spaces. Most of the functions here are *vectorized*, allowing for array inputs to convert multiple color values at once.

As of version 2021.0 of PsychoPy, users ought to use the *Color* class for working with color coordinate values.

CIELAB

Conversion functions for the *CIELAB* and *CIELCH* color space.

<code>cielab2rgb(lab[, whiteXYZ, ...])</code>	Transform CIE L*a*b* (1976) color space coordinates to RGB tristimulus values.
<code>cielch2rgb(lch[, whiteXYZ, ...])</code>	Transform CIE L*C*h* coordinates to RGB tristimulus values.

psychopy.tools.colorspectools.cielab2rgb

`psychopy.tools.colorspectools.cielab2rgb(lab, whiteXYZ=None, conversionMatrix=None, transferFunc=None, clip=False, **kwargs)`

Transform CIE L*a*b* (1976) color space coordinates to RGB tristimulus values.

CIE L*a*b* are first transformed into CIE XYZ (1931) color space, then the RGB conversion is applied. By default, the sRGB conversion matrix is used with a reference D65 white point. You may specify your own RGB conversion matrix and white point (in CIE XYZ) appropriate for your display.

Parameters

- **lab** (*tuple, list or ndarray*) – 1-, 2-, 3-D vector of CIE L*a*b* coordinates to convert. The last dimension should be length-3 in all cases specifying a single coordinate.
- **whiteXYZ** (*tuple, list or ndarray*) – 1-D vector coordinate of the white point in CIE-XYZ color space. Must be the same white point needed by the conversion matrix. The default white point is D65 if None is specified, defined as X, Y, Z = 0.9505, 1.0000, 1.0890.
- **conversionMatrix** (*tuple, list or ndarray*) – 3x3 conversion matrix to transform CIE-XYZ to RGB values. The default matrix is sRGB with a D65 white point if None is specified. Note that values must be gamma corrected to appear correctly according to the sRGB standard.
- **transferFunc** (*pyfunc or None*) – Signature of the transfer function to use. If None, values are kept as linear RGB (it's assumed your display is gamma corrected via the hardware CLUT). The TF must be appropriate for the conversion matrix supplied (default is sRGB). Additional arguments to 'transferFunc' can be passed by specifying them as keyword arguments. Gamma functions that come with PsychoPy are 'srgbTF' and 'rec709TF', see their docs for more information.

- **clip** (*bool*) – Make all output values representable by the display. However, colors outside of the display’s gamut may not be valid!

Returns Array of RGB tristimulus values.

Return type ndarray

Example

Converting a CIE $L^*a^*b^*$ color to linear RGB:

```
import psychopy.tools.colorspectools as cst
cielabColor = (53.0, -20.0, 0.0) # greenish color (L*, a*, b*)
rgbColor = cst.cielab2rgb(cielabColor)
```

Using a transfer function to convert to sRGB:

```
rgbColor = cst.cielab2rgb(cielabColor, transferFunc=cst.srgbTF)
```

psychopy.tools.colorspectools.cielch2rgb

psychopy.tools.colorspectools.**cielch2rgb** (*lch*, *whiteXYZ=None*, *conversionMatrix=None*, *transferFunc=None*, *clip=False*, ***kwargs*)

Transform CIE $L^*C^*h^*$ coordinates to RGB tristimulus values.

Parameters

- **lch** (*tuple*, *list* or *ndarray*) – 1-, 2-, 3-D vector of CIE $L^*C^*h^*$ coordinates to convert. The last dimension should be length-3 in all cases specifying a single coordinate. The hue angle *h* is expected in degrees.
- **whiteXYZ** (*tuple*, *list* or *ndarray*) – 1-D vector coordinate of the white point in CIE-XYZ color space. Must be the same white point needed by the conversion matrix. The default white point is D65 if None is specified, defined as X, Y, Z = 0.9505, 1.0000, 1.0890
- **conversionMatrix** (*tuple*, *list* or *ndarray*) – 3x3 conversion matrix to transform CIE-XYZ to RGB values. The default matrix is sRGB with a D65 white point if None is specified. Note that values must be gamma corrected to appear correctly according to the sRGB standard.
- **transferFunc** (*pyfunc* or *None*) – Signature of the transfer function to use. If None, values are kept as linear RGB (it’s assumed your display is gamma corrected via the hardware CLUT). The TF must be appropriate for the conversion matrix supplied. Additional arguments to ‘transferFunc’ can be passed by specifying them as keyword arguments. Gamma functions that come with PsychoPy are ‘srgbTF’ and ‘rec709TF’, see their docs for more information.
- **clip** (*boolean*) – Make all output values representable by the display. However, colors outside of the display’s gamut may not be valid!

Returns array of RGB tristimulus values

Return type ndarray

DKL

Conversion functions for the *Derrington Krauskopf and Lennie* (DKL) color space.

<code>dkl2rgb(dkl[, conversionMatrix])</code>	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.
<code>dklCart2rgb(LUM, LM, S[, conversionMatrix])</code>	Like <code>dkl2rgb</code> except that it uses cartesian coords (LM,S,LUM) rather than spherical coords for DKL (elev, azim, contr).
<code>rgb2dklCart(picture[, conversionMatrix])</code>	Convert an RGB image into Cartesian DKL space.

psychopy.tools.colorspacetools.dkl2rgb

`psychopy.tools.colorspacetools.dkl2rgb` (*dkl, conversionMatrix=None*)

Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that this will not be an accurate representation of the color space unless you supply a conversion matrix).

Examples

Converting a single DKL color to RGB:

```
dkl = [90, 0, 1]
rgb = dkl2rgb(dkl, conversionMatrix)
```

psychopy.tools.colorspacetools.dklCart2rgb

`psychopy.tools.colorspacetools.dklCart2rgb` (*LUM, LM, S, conversionMatrix=None*)

Like `dkl2rgb` except that it uses cartesian coords (LM,S,LUM) rather than spherical coords for DKL (elev, azim, contr).

NB: this may return rgb values >1 or <-1

psychopy.tools.colorspacetools.rgb2dklCart

`psychopy.tools.colorspacetools.rgb2dklCart` (*picture, conversionMatrix=None*)

Convert an RGB image into Cartesian DKL space.

HSV

Conversion functions for the *Hue-Saturation-Value* (HSV) color space.

<code>hsv2rgb(hsv_Nx3)</code>	Convert from HSV color space to RGB gun values.
<code>rgb2hsv(rgb)</code>	Convert values from linear RGB to HSV colorspace.

psychoPy.tools.colorspacetools.hsv2rgb

psychoPy.tools.colorspacetools.**hsv2rgb** (*hsv_Nx3*)

Convert from HSV color space to RGB gun values.

usage:

```
rgb_Nx3 = hsv2rgb(hsv_Nx3)
```

Note that in some uses of HSV space the Hue component is given in radians or cycles (range 0:1]). In this version H is given in degrees (0:360).

Also note that the RGB output ranges -1:1, in keeping with other PsychoPy functions.

psychoPy.tools.colorspacetools.rgb2hsv

psychoPy.tools.colorspacetools.**rgb2hsv** (*rgb*)

Convert values from linear RGB to HSV colorspace.

Parameters **rgb** (*array_like*) – 1-, 2-, 3-D vector of RGB coordinates to convert. The last dimension should be length-3 in all cases, specifying a single coordinate.

Returns HSV values with the same shape as the input.

Return type ndarray

LMS

<code>lms2rgb(lms_Nx3[, conversionMatrix])</code>	Convert from cone space (Long, Medium, Short) to RGB.
<code>rgb2lms(rgb_Nx3[, conversionMatrix])</code>	Convert from RGB to cone space (LMS).

psychoPy.tools.colorspacetools.lms2rgb

psychoPy.tools.colorspacetools.**lms2rgb** (*lms_Nx3, conversionMatrix=None*)

Convert from cone space (Long, Medium, Short) to RGB.

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that you will not get an accurate representation of the color space unless you supply a conversion matrix)

usage:

```
rgb_Nx3 = lms2rgb(dkl_Nx3(e1, az, radius), conversionMatrix)
```

psychoPy.tools.colorspectools.rgb2lms

psychoPy.tools.colorspectools.**rgb2lms** (*rgb_Nx3*, *conversionMatrix=None*)

Convert from RGB to cone space (LMS).

Requires a conversion matrix, which will be generated from generic Sony Trinitron phosphors if not supplied (note that you will not get an accurate representation of the color space unless you supply a conversion matrix)

usage:

```
lms_Nx3 = rgb2lms(rgb_Nx3(e1, az, radius), conversionMatrix)
```

Gamma/Transfer Functions

Standard gamma functions for converting between *linear RGB* space and *sRGB*.

<code>srgbTF(rgb[, reverse])</code>	Apply sRGB transfer function (or gamma) to linear RGB values.
<code>rec709TF(rgb, **kwargs)</code>	Apply the Rec 709 transfer function (or gamma) to linear RGB values.

psychoPy.tools.colorspectools.srgbTF

psychoPy.tools.colorspectools.**srgbTF** (*rgb*, *reverse=False*, ***kwargs*)

Apply sRGB transfer function (or gamma) to linear RGB values.

Input values must have been transformed using a conversion matrix derived from sRGB primaries relative to D65.

Parameters

- **rgb** (*tuple*, *list* or *ndarray of floats*) – Nx3 or NxNx3 array of linear RGB values, last dim must be size == 3 specifying RBG values.
- **reverse** (*boolean*) – If True, the reverse transfer function will convert sRGB -> linear RGB.

Returns Array of transformed colors with same shape as input.

Return type ndarray

psychoPy.tools.colorspectools.rec709TF

psychoPy.tools.colorspectools.**rec709TF** (*rgb*, ***kwargs*)

Apply the Rec 709 transfer function (or gamma) to linear RGB values.

This transfer function is defined in the ITU-R BT.709 (2015) recommendation document (<https://www.itu.int/rec/R-REC-BT.709-6-201506-I/en>) and is commonly used with HDTV televisions.

Parameters **rgb** (*tuple*, *list* or *ndarray of floats*) – Nx3 or NxNx3 array of linear RGB values, last dim must be size == 3 specifying RBG values.

Returns Array of transformed colors with same shape as input.

Return type ndarray

Helpers

Helper functions for working with color coordinates.

<code>rescaleColor(rgb[, convertTo, clip])</code>	Rescale RGB colors.
---	---------------------

psychoPy.tools.colorspectools.rescaleColor

`psychoPy.tools.colorspectools.rescaleColor` (*rgb*, *convertTo*='signed', *clip*=False)

Rescale RGB colors.

This function can be used to convert RGB value triplets from the PsychoPy signed color format to the normalized OpenGL format.

PsychoPy represents colors using values between -1 and 1. However, colors are commonly represented using values between 0 and 1 when working with OpenGL and various other contexts. This function simply rescales values to switch between these formats.

Parameters

- **rgb** (*array_like*) – 1-, 2-, 3-D vector of RGB coordinates to convert. The last dimension should be length-3 in all cases, specifying a single coordinate.
- **convertTo** (*str*) – If 'signed', this function will assume *rgb* is in OpenGL format [0:1] and rescale them to PsychoPy's format [-1:1]. If 'unsigned', input values are treated as OpenGL format and will be rescaled to use PsychoPy's. Default is 'signed'.
- **clip** (*bool*) – Clip values to the range that can be represented on a display. This is an optional step. Default is *False*.

Returns Rescaled values with the same shape as *rgb*.

Return type ndarray

Notes

The *convertTo* argument also accepts strings 'opengl' and 'psychoPy' as substitutes for 'signed' and 'unsigned', respectively. This might be more explicit in some contexts.

Examples

Convert a signed RGB value to unsigned format:

```
rgb_signed = [-1, 0, 1]
rgb_unsigned = rescaleColor(rgb_signed, convertTo='unsigned')
```

9.7.2 psychopy.tools.coordinatetools

Functions and classes related to coordinate system conversion

<code>cart2pol(x, y[, units])</code>	Convert from cartesian to polar coordinates.
<code>cart2sph(z, y, x)</code>	Convert from cartesian coordinates (x,y,z) to spherical (elevation, azimuth, radius).
<code>pol2cart(theta, radius[, units])</code>	Convert from polar to cartesian coordinates.
<code>sph2cart(*args)</code>	Convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x,y,z).

Function details

`psychopy.tools.coordinatetools.cart2pol(x, y, units='deg')`

Convert from cartesian to polar coordinates.

Usage `theta, radius = cart2pol(x, y, units='deg')`

`units` refers to the units (rad or deg) for `theta` that should be returned

`psychopy.tools.coordinatetools.cart2sph(z, y, x)`

Convert from cartesian coordinates (x,y,z) to spherical (elevation, azimuth, radius). Output is in degrees.

usage: `array3xN[el,az,rad] = cart2sph(array3xN[x,y,z])` OR `elevation, azimuth, radius = cart2sph(x,y,z)`

If working in DKL space, `z = Luminance`, `y = S` and `x = LM`

`psychopy.tools.coordinatetools.pol2cart(theta, radius, units='deg')`

Convert from polar to cartesian coordinates.

usage:

```
x, y = pol2cart(theta, radius, units='deg')
```

`psychopy.tools.coordinatetools.sph2cart(*args)`

Convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x,y,z).

usage: `array3xN[x,y,z] = sph2cart(array3xN[el,az,rad])` OR `x,y,z = sph2cart(elev, azim, radius)`

9.7.3 psychopy.tools.filetools

Functions and classes related to file and directory handling

`psychopy.tools.filetools.toFile(filename, data)`

Save data (of any sort) as a pickle file.

simple wrapper of the `cPickle` module in core python

`psychopy.tools.filetools.fromFile(filename, encoding='utf-8-sig')`

Load data from a psydat, pickle or JSON file.

Parameters `encoding (str)` – The encoding to use when reading a JSON file. This parameter will be ignored for any other file type.

`psychopy.tools.filetools.mergeFolder(src, dst, pattern=None)`

Merge a folder into another.

Existing files in `dst` folder with the same name will be overwritten. Non-existent files/folders will be created.

`psychoPy.tools.filetools.openOutputFile` (*fileName=None, append=False, fileCollisionMethod='rename', encoding='utf-8-sig'*)

Open an output file (or standard output) for writing.

Parameters

fileName [None, 'stdout', or str] The desired output file name. If *None* or *stdout*, return *sys.stdout*. Any other string will be considered a filename.

append [bool, optional] If *True*, append data to an existing file; otherwise, overwrite it with new data. Defaults to *True*, i.e. appending.

fileCollisionMethod [string, optional] How to handle filename collisions. Valid values are 'rename', 'overwrite', and 'fail'. This parameter is ignored if *append* is set to *True*. Defaults to *rename*.

encoding [string, optional] The encoding to use when writing the file. This parameter will be ignored if *append* is *False* and *fileName* ends with *.psydat* or *.npy* (i.e. if a binary file is to be written). Defaults to 'utf-8'.

Returns

f [file] A writable file handle.

`psychoPy.tools.filetools.genDelimiter` (*fileName*)

Return a delimiter based on a filename.

Parameters

fileName [string] The output file name.

Returns

delim [string] A delimiter picked based on the supplied filename. This will be *,* if the filename extension is *.csv*, and a tabulator character otherwise.

9.7.4 psychoPy.tools.gltools

OpenGL related helper functions.

Shaders

Tools for creating, compiling, using, and inspecting shader programs.

<code>createProgram()</code>	Create an empty program object for shaders.
<code>createProgramObjectARB()</code>	Create an empty program object for shaders.
<code>compileShader(shaderSrc, shaderType)</code>	Compile shader GLSL code and return a shader object.
<code>compileShaderObjectARB(shaderSrc, shaderType)</code>	Compile shader GLSL code and return a shader object.
<code>embedShaderSourceDefs(shaderSrc, defs)</code>	Embed preprocessor definitions into GLSL source code.
<code>deleteObject(obj)</code>	Delete a shader or program object.
<code>deleteObjectARB(obj)</code>	Delete a program or shader object.
<code>attachShader(program, shader)</code>	Attach a shader to a program.
<code>attachObjectARB(program, shader)</code>	Attach a shader object to a program.
<code>detachShader(program, shader)</code>	Detach a shader object from a program.

continues on next page

Table 9.51 – continued from previous page

<code>detachObjectARB(program, shader)</code>	Detach a shader object from a program.
<code>linkProgram(program)</code>	Link a shader program.
<code>linkProgramObjectARB(program)</code>	Link a shader program object.
<code>validateProgram(program)</code>	Check if the program can execute given the current OpenGL state.
<code>validateProgramARB(program)</code>	Check if the program can execute given the current OpenGL state.
<code>useProgram(program)</code>	Use a program object's executable shader attachments in the current OpenGL rendering state.
<code>useProgramObjectARB(program)</code>	Use a program object's executable shader attachments in the current OpenGL rendering state.
<code>getInfoLog(obj)</code>	Get the information log from a shader or program.
<code>getUniformLocations(program[, builtins])</code>	Get uniform names and locations from a given shader program object.
<code>getAttribLocations(program[, builtins])</code>	Get attribute names and locations from the specified program object.

psychopy.tools.gltools.createProgram

`psychopy.tools.gltools.createProgram()`

Create an empty program object for shaders.

Returns OpenGL program object handle retrieved from a `glCreateProgram` call.

Return type `int`

Examples

Building a program with vertex and fragment shader attachments:

```
myProgram = createProgram() # new shader object

# compile vertex and fragment shader sources
vertexShader = compileShader(vertexShaderSource, GL.GL_VERTEX_SHADER)
fragmentShader = compileShader(fragmentShaderSource, GL.GL_FRAGMENT_SHADER)

# attach shaders to program
attachShader(myProgram, vertexShader)
attachShader(myProgram, fragmentShader)

# link the shader, makes `myProgram` attachments executable by their
# respective processors and available for use
linkProgram(myProgram)

# optional, validate the program
validateProgram(myProgram)

# optional, detach and discard shader objects
detachShader(myProgram, vertexShader)
detachShader(myProgram, fragmentShader)

deleteObject(vertexShader)
deleteObject(fragmentShader)
```

You can install the program for use in the current rendering state by calling:

```
useProgram(myShader) # OR glUseProgram(myShader)
# set uniforms/attributes and start drawing here ...
```

psychopy.tools.gltools.createProgramObjectARB

psychopy.tools.gltools.createProgramObjectARB()

Create an empty program object for shaders.

This creates an *Architecture Review Board* (ARB) program variant which is compatible with older GLSL versions and OpenGL coding practices (eg. immediate mode) on some platforms. Use **ARB* variants of shader helper functions (eg. *compileShaderObjectARB* instead of *compileShader*) when working with these ARB program objects. This was included for legacy support of existing PsychoPy shaders. However, it is recommended that you use *createShader()* and follow more recent OpenGL design patterns for new code (if possible of course).

Returns OpenGL program object handle retrieved from a *glCreateProgramObjectARB* call.

Return type int

Examples

Building a program with vertex and fragment shader attachments:

```
myProgram = createProgramObjectARB() # new shader object

# compile vertex and fragment shader sources
vertexShader = compileShaderObjectARB(
    vertShaderSource, GL.GL_VERTEX_SHADER_ARB)
fragmentShader = compileShaderObjectARB(
    fragShaderSource, GL.GL_FRAGMENT_SHADER_ARB)

# attach shaders to program
attachObjectARB(myProgram, vertexShader)
attachObjectARB(myProgram, fragmentShader)

# link the shader, makes `myProgram` attachments executable by their
# respective processors and available for use
linkProgramObjectARB(myProgram)

# optional, validate the program
validateProgramARB(myProgram)

# optional, detach and discard shader objects
detachObjectARB(myProgram, vertexShader)
detachObjectARB(myProgram, fragmentShader)

deleteObjectARB(vertexShader)
deleteObjectARB(fragmentShader)
```

Use the program in the current OpenGL state:

```
useProgramObjectARB(myProgram)
```

psychopy.tools.gltools.compileShader

psychopy.tools.gltools.**compileShader** (*shaderSrc*, *shaderType*)

Compile shader GLSL code and return a shader object. Shader objects can then be attached to programs and made executable on their respective processors.

Parameters

- **shaderSrc** (*str*, *list of str*) – GLSL shader source code.
- **shaderType** (*GLenum*) – Shader program type (eg. `GL_VERTEX_SHADER`, `GL_FRAGMENT_SHADER`, `GL_GEOMETRY_SHADER`, etc.)

Returns OpenGL shader object handle retrieved from a `glCreateShader` call.

Return type `int`

Examples

Compiling GLSL source code and attaching it to a program object:

```
# GLSL vertex shader source
vertexSource = '''
    #version 330 core
    layout (location = 0) in vec3 vertexPos;

    void main()
    {
        gl_Position = vec4(vertexPos, 1.0);
    }
'''

# compile it, specifying `GL_VERTEX_SHADER`
vertexShader = compileShader(vertexSource, GL.GL_VERTEX_SHADER)
attachShader(myProgram, vertexShader) # attach it to `myProgram`
```

psychopy.tools.gltools.compileShaderObjectARB

psychopy.tools.gltools.**compileShaderObjectARB** (*shaderSrc*, *shaderType*)

Compile shader GLSL code and return a shader object. Shader objects can then be attached to programs and made executable on their respective processors.

Parameters

- **shaderSrc** (*str*, *list of str*) – GLSL shader source code text.
- **shaderType** (*GLenum*) – Shader program type. Must be `*_ARB` enums such as `GL_VERTEX_SHADER_ARB`, `GL_FRAGMENT_SHADER_ARB`, `GL_GEOMETRY_SHADER_ARB`, etc.

Returns OpenGL shader object handle retrieved from a `glCreateShaderObjectARB` call.

Return type `int`

psychopy.tools.gltools.embedShaderSourceDefs

psychopy.tools.gltools.**embedShaderSourceDefs** (*shaderSrc*, *defs*)

Embed preprocessor definitions into GLSL source code.

This function generates and inserts `#define` statements into existing GLSL source code, allowing one to use GLSL preprocessor statements to alter program source at compile time.

Passing `{'MAX_LIGHTS': 8, 'NORMAL_MAP': False}` to *defs* will create and insert the following `#define` statements into *shaderSrc*:

```
#define MAX_LIGHTS 8
#define NORMAL_MAP 0
```

As per the GLSL specification, the `#version` directive must be specified at the top of the file before any other statement (with the exception of comments). If a `#version` directive is present, generated `#define` statements will be inserted starting at the following line. If no `#version` directive is found in *shaderSrc*, the statements will be prepended to *shaderSrc*.

Using preprocessor directives, multiple shader program routines can reside in the same source text if enclosed by `#ifdef` and `#endif` statements as shown here:

```
#ifdef VERTEX
    // vertex shader code here ...
#endif

#ifdef FRAGMENT
    // pixel shader code here ...
#endif
```

Both the vertex and fragment shader can be built from the same GLSL code listing by setting either `VERTEX` or `FRAGMENT` as *True*:

```
vertexShader = gltools.compileShaderObjectARB(
    gltools.embedShaderSourceDefs(glslSource, {'VERTEX': True}),
    GL.GL_VERTEX_SHADER_ARB)
fragmentShader = gltools.compileShaderObjectARB(
    gltools.embedShaderSourceDefs(glslSource, {'FRAGMENT': True}),
    GL.GL_FRAGMENT_SHADER_ARB)
```

In addition, `#ifdef` blocks can be used to prune render code paths. Here, this GLSL snippet shows a shader having diffuse color sampled from a texture is conditional on `DIFFUSE_TEXTURE` being *True*, if not, the material color is used instead:

```
#ifdef DIFFUSE_TEXTURE
    uniform sampler2D diffuseTexture;
#endif
...
#ifdef DIFFUSE_TEXTURE
    // sample color from texture
    vec4 diffuseColor = texture2D(diffuseTexture, gl_TexCoord[0].st);
#else
    // code path for no textures, just output material color
    vec4 diffuseColor = gl_FrontMaterial.diffuse;
#endif
```

This avoids needing to provide two separate GLSL program sources to build shaders to handle cases where a diffuse texture is or isn't used.

Parameters

- **shaderSrc** (*str*) – GLSL shader source code.
- **defs** (*dict*) – Names and values to generate `#define` statements. Keys must all be valid GLSL preprocessor variable names of type *str*. Values can only be *int*, *float*, *str*, *bytes*, or *bool* types. Boolean values *True* and *False* are converted to integers *1* and *0*, respectively.

Returns GLSL source code with `#define` statements inserted.

Return type *str*

Examples

Defining `MAX_LIGHTS` as `8` in a fragment shader program at runtime:

```
fragSrc = embedShaderSourceDefs(fragSrc, {'MAX_LIGHTS': 8})
fragShader = compileShaderObjectARB(fragSrc, GL_FRAGMENT_SHADER_ARB)
```

`psychopy.tools.gltools.deleteObject`

`psychopy.tools.gltools.deleteObject` (*obj*)

Delete a shader or program object.

Parameters *obj* (*int*) – Shader or program object handle. Must have originated from a `createProgram()`, `compileShader()`, `glCreateProgram` or `glCreateShader` call.

`psychopy.tools.gltools.deleteObjectARB`

`psychopy.tools.gltools.deleteObjectARB` (*obj*)

Delete a program or shader object.

Parameters *obj* (*int*) – Program handle to attach *shader* to. Must have originated from a `createProgramObjectARB()`, `compileShaderObjectARB()`, `glCreateProgramObjectARB()` or `glCreateShaderObjectARB` call.

`psychopy.tools.gltools.attachShader`

`psychopy.tools.gltools.attachShader` (*program*, *shader*)

Attach a shader to a program.

Parameters

- **program** (*int*) – Program handle to attach *shader* to. Must have originated from a `createProgram()` or `glCreateProgram` call.
- **shader** (*int*) – Handle of shader object to attach. Must have originated from a `compileShader()` or `glCreateShader` call.

psychopy.tools.gltools.attachObjectARB

psychopy.tools.gltools.**attachObjectARB** (*program*, *shader*)

Attach a shader object to a program.

Parameters

- **program** (*int*) – Program handle to attach *shader* to. Must have originated from a `createProgramObjectARB()` or `glCreateProgramObjectARB` call.
- **shader** (*int*) – Handle of shader object to attach. Must have originated from a `compileShaderObjectARB()` or `glCreateShaderObjectARB` call.

psychopy.tools.gltools.detachShader

psychopy.tools.gltools.**detachShader** (*program*, *shader*)

Detach a shader object from a program.

Parameters

- **program** (*int*) – Program handle to detach *shader* from. Must have originated from a `createProgram()` or `glCreateProgram` call.
- **shader** (*int*) – Handle of shader object to detach. Must have been previously attached to *program*.

psychopy.tools.gltools.detachObjectARB

psychopy.tools.gltools.**detachObjectARB** (*program*, *shader*)

Detach a shader object from a program.

Parameters

- **program** (*int*) – Program handle to detach *shader* from. Must have originated from a `createProgramObjectARB()` or `glCreateProgramObjectARB` call.
- **shader** (*int*) – Handle of shader object to detach. Must have been previously attached to *program*.

psychopy.tools.gltools.linkProgram

psychopy.tools.gltools.**linkProgram** (*program*)

Link a shader program. Any attached shader objects will be made executable to run on associated GPU processor units when the program is used.

Parameters **program** (*int*) – Program handle to link. Must have originated from a `createProgram()` or `glCreateProgram` call.

Raises

- **ValueError** – Specified *program* handle is invalid.
- **RuntimeError** – Program failed to link. Log will be dumped to *stderr*.

psychopy.tools.gltools.linkProgramObjectARB

`psychopy.tools.gltools.linkProgramObjectARB(program)`

Link a shader program object. Any attached shader objects will be made executable to run on associated GPU processor units when the program is used.

Parameters `program` (*int*) – Program handle to link. Must have originated from a `createProgramObjectARB()` or `glCreateProgramObjectARB` call.

Raises

- **ValueError** – Specified *program* handle is invalid.
- **RuntimeError** – Program failed to link. Log will be dumped to *stderr*.

psychopy.tools.gltools.validateProgram

`psychopy.tools.gltools.validateProgram(program)`

Check if the program can execute given the current OpenGL state.

Parameters `program` (*int*) – Handle of program to validate. Must have originated from a `createProgram()` or `glCreateProgram` call.

psychopy.tools.gltools.validateProgramARB

`psychopy.tools.gltools.validateProgramARB(program)`

Check if the program can execute given the current OpenGL state. If validation fails, information from the driver is dumped giving the reason.

Parameters `program` (*int*) – Handle of program object to validate. Must have originated from a `createProgramObjectARB()` or `glCreateProgramObjectARB` call.

psychopy.tools.gltools.useProgram

`psychopy.tools.gltools.useProgram(program)`

Use a program object's executable shader attachments in the current OpenGL rendering state.

In order to install the program object in the current rendering state, a program must have been successfully linked by calling `linkProgram()` or `glLinkProgram`.

Parameters `program` (*int*) – Handle of program to use. Must have originated from a `createProgram()` or `glCreateProgram` call and was successfully linked. Passing `0` or `None` disables shader programs.

Examples

Install a program for use in the current rendering state:

```
useProgram(myShader)
```

Disable the current shader program by specifying `0`:

```
useProgram(0)
```

psychopy.tools.gltools.useProgramObjectARB

`psychopy.tools.gltools.useProgramObjectARB(program)`

Use a program object's executable shader attachments in the current OpenGL rendering state.

In order to install the program object in the current rendering state, a program must have been successfully linked by calling `linkProgramObjectARB()` or `glLinkProgramObjectARB`.

Parameters `program` (*int*) – Handle of program object to use. Must have originated from a `createProgramObjectARB()` or `glCreateProgramObjectARB` call and was successfully linked. Passing `0` or `None` disables shader programs.

Examples

Install a program for use in the current rendering state:

```
useProgramObjectARB(myShader)
```

Disable the current shader program by specifying `0`:

```
useProgramObjectARB(0)
```

Notes

Some drivers may support using `glUseProgram` for objects created by calling `createProgramObjectARB()` or `glCreateProgramObjectARB`.

psychopy.tools.gltools.getInfoLog

`psychopy.tools.gltools.getInfoLog(obj)`

Get the information log from a shader or program.

This retrieves a text log from the driver pertaining to the shader or program. For instance, a log can report shader compiler output or validation results. The verbosity and formatting of the logs are platform-dependent, where one driver may provide more information than another.

This function works with both standard and ARB program object variants.

Parameters `obj` (*int*) – Program or shader to retrieve a log from. If a shader, the handle must have originated from a `compileShader()`, `glCreateShader`, `createProgramObjectARB()` or `glCreateProgramObjectARB` call. If a program, the handle must have come from a `createProgram()`, `createProgramObjectARB()`, `glCreateProgram` or `glCreateProgramObjectARB` call.

Returns Information log data. Logs can be empty strings if the driver has no information available.

Return type `str`

psychopy.tools.gltools.getUniformLocations

`psychopy.tools.gltools.getUniformLocations` (*program*, *builtins=False*)

Get uniform names and locations from a given shader program object.

This function works with both standard and ARB program object variants.

Parameters

- **program** (*int*) – Handle of program to retrieve uniforms. Must have originated from a `createProgram()`, `createProgramObjectARB()`, `glCreateProgram` or `glCreateProgramObjectARB` call.
- **builtins** (*bool*, *optional*) – Include built-in GLSL uniforms (eg. `gl_ModelViewProjectionMatrix`). Default is *False*.

Returns Uniform names and locations.

Return type `dict`

psychopy.tools.gltools.getAttribLocations

`psychopy.tools.gltools.getAttribLocations` (*program*, *builtins=False*)

Get attribute names and locations from the specified program object.

This function works with both standard and ARB program object variants.

Parameters

- **program** (*int*) – Handle of program to retrieve attributes. Must have originated from a `createProgram()`, `createProgramObjectARB()`, `glCreateProgram` or `glCreateProgramObjectARB` call.
- **builtins** (*bool*, *optional*) – Include built-in GLSL attributes (eg. `gl_Vertex`). Default is *False*.

Returns Attribute names and locations.

Return type `dict`

Query

Tools for using OpenGL query objects.

<code>createQueryObject([target])</code>	Create a GL query object.
<code>QueryObjectInfo(name, target)</code>	Object for querying information.
<code>beginQuery(query)</code>	Begin query.
<code>endQuery(query)</code>	End a query.
<code>getQuery(query)</code>	Get the value stored in a query object.
<code>getAbsTimeGPU()</code>	Get the absolute GPU time in nanoseconds.

psychoPy.tools.gltools.createQueryObject

psychoPy.tools.gltools.**createQueryObject** (*target=35007*)

Create a GL query object.

Parameters **target** (*GLenum or int*) – Target for the query.

Returns Query object.

Return type *QueryObjectInfo*

Examples

Get GPU time elapsed executing rendering/GL calls associated with some stimuli (this is not the difference in absolute time between consecutive *beginQuery* and *endQuery* calls!):

```
# create a new query object
qGPU = createQueryObject(GL_TIME_ELAPSED)

beginQuery(query)
myStim.draw() # OpenGL calls here
endQuery(query)

# get time elapsed in seconds spent on the GPU
timeRendering = getQueryValue(qGPU) * 1e-9
```

You can also use queries to test if vertices are occluded, as their samples would be rejected during depth testing:

```
drawVAO(shape0, GL_TRIANGLES) # draw the first object

# check if the object was completely occluded
qOcclusion = createQueryObject(GL_ANY_SAMPLES_PASSED)

# draw the next shape within query context
beginQuery(qOcclusion)
drawVAO(shape1, GL_TRIANGLES) # draw the second object
endQuery(qOcclusion)

isOccluded = getQueryValue(qOcclusion) == 1
```

This can be leveraged to perform occlusion testing/culling, where you can render a *cheap* version of your mesh/shape, then the more expensive version if samples were passed.

psychoPy.tools.gltools.QueryObjectInfo

class psychoPy.tools.gltools.**QueryObjectInfo** (*name, target*)

Object for querying information. This includes GPU timing information.

__init__ (*name, target*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(name, target)</code>	Initialize self.
<code>isValid()</code>	Check if the name associated with this object is valid.

Attributes

<code>name</code>
<code>target</code>

`psychopy.tools.gltools.beginQuery`

`psychopy.tools.gltools.beginQuery(query)`
Begin query.

Parameters `query` (`QueryObjectInfo`) – Query object descriptor returned by `createQueryObject()`.

`psychopy.tools.gltools.endQuery`

`psychopy.tools.gltools.endQuery(query)`
End a query.

Parameters `query` (`QueryObjectInfo`) – Query object descriptor returned by `createQueryObject()`, previously passed to `beginQuery()`.

`psychopy.tools.gltools.getQuery`

`psychopy.tools.gltools.getQuery(query)`
Get the value stored in a query object.

Parameters `query` (`QueryObjectInfo`) – Query object descriptor returned by `createQueryObject()`, previously passed to `endQuery()`.

`psychopy.tools.gltools.getAbsTimeGPU`

`psychopy.tools.gltools.getAbsTimeGPU()`
Get the absolute GPU time in nanoseconds.

Returns Time elapsed in nanoseconds since the OpenGL context was fully realized.

Return type `int`

Examples

Get the current GPU time in seconds:

```
timeInSeconds = getAbsTimeGPU() * 1e-9
```

Get the GPU time elapsed:

```
t0 = getAbsTimeGPU()
# some drawing commands here ...
t1 = getAbsTimeGPU()
timeElapsed = (t1 - t0) * 1e-9 # take difference, convert to seconds
```

Framebuffer Objects (FBO)

Tools for creating Framebuffer Objects (FBOs).

<code>createFBO([attachments])</code>	Create a Framebuffer Object.
<code>attach(attachPoint, imageBuffer)</code>	Attach an image to a specified attachment point on the presently bound FBO.
<code>isComplete()</code>	Check if the currently bound framebuffer is complete.
<code>deleteFBO(fbo)</code>	Delete a framebuffer.
<code>blitFBO(srcRect[, dstRect, filter])</code>	Copy a block of pixels between framebuffers via blitting.
<code>useFBO(fbo)</code>	Context manager for Framebuffer Object bindings.

psychoPy.tools.gltools.createFBO

`psychoPy.tools.gltools.createFBO(attachments=())`

Create a Framebuffer Object.

Parameters `attachments` (list or tuple of tuple) – Optional attachments to initialize the Framebuffer with. Attachments are specified as a list of tuples. Each tuple must contain an attachment point (e.g. `GL_COLOR_ATTACHMENT0`, `GL_DEPTH_ATTACHMENT`, etc.) and a buffer descriptor type (Renderbuffer or `TexImage2D`). If using a combined depth/stencil format such as `GL_DEPTH24_STENCIL8`, `GL_DEPTH_ATTACHMENT` and `GL_STENCIL_ATTACHMENT` must be passed the same buffer. Alternatively, one can use `GL_DEPTH_STENCIL_ATTACHMENT` instead. If using multisample buffers, all attachment images must use the same number of samples!. As an example, one may specify attachments as `'attachments=((GL.GL_COLOR_ATTACHMENT0, frameTexture), (GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRenderBuffer))'`.

Returns Framebuffer descriptor.

Return type Framebuffer

Notes

- All buffers must have the same number of samples.
- The 'userData' field of the returned descriptor is a dictionary that can be used to store arbitrary data associated with the FBO.
- Framebuffers need a single attachment to be complete.

Examples

Create an empty framebuffer with no attachments:

```
fbo = createFBO() # invalid until attachments are added
```

Create a render target with multiple color texture attachments:

```
colorTex = createTexImage2D(1024,1024) # empty texture
depthRb = createRenderbuffer(800,600,internalFormat=GL.GL_DEPTH24_STENCIL8)

# attach images
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fbo.id)
attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
attach(GL.GL_STENCIL_ATTACHMENT, depthRb)
# or attach(GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRb)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, 0)

# above is the same as
with useFBO(fbo):
    attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
    attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
    attach(GL.GL_STENCIL_ATTACHMENT, depthRb)
```

Examples of userData some custom function might access:

```
fbo.userData['flags'] = ['left_eye', 'clear_before_use']
```

Using a depth only texture (for shadow mapping?):

```
depthTex = createTexImage2D(800, 600,
                             internalFormat=GL.GL_DEPTH_COMPONENT24,
                             pixelFormat=GL.GL_DEPTH_COMPONENT)
fbo = createFBO([(GL.GL_DEPTH_ATTACHMENT, depthTex)]) # is valid

# discard FBO descriptor, just give me the ID
frameBuffer = createFBO().id
```

psychopy.tools.gltools.attach

`psychopy.tools.gltools.attach(attachPoint, imageBuffer)`

Attach an image to a specified attachment point on the presently bound FBO.

:param attachPoint `int`: Attachment point for 'imageBuffer' (e.g. `GL.GL_COLOR_ATTACHMENT0`).
 :param imageBuffer: Framebuffer-attachable buffer descriptor. :type imageBuffer: `TexImage2D` or `Renderbuffer`

Examples

Attach an image to attachment points on the framebuffer:

```
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fbo)
attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
attach(GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRb)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, lastBoundFbo)

# same as above, but using a context manager
with useFBO(fbo):
    attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
    attach(GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRb)
```

psychopy.tools.gltools.isComplete

`psychopy.tools.gltools.isComplete()`

Check if the currently bound framebuffer is complete.

Returns `True` if the presently bound FBO is complete.

Return type `bool`

psychopy.tools.gltools.deleteFBO

`psychopy.tools.gltools.deleteFBO(fbo)`

Delete a framebuffer.

psychopy.tools.gltools.blitFBO

`psychopy.tools.gltools.blitFBO(srcRect, dstRect=None, filter=9729)`

Copy a block of pixels between framebuffers via blitting. Read and draw framebuffers must be bound prior to calling this function. Beware, the scissor box and viewport are changed when this is called to `dstRect`.

Parameters

- **srcRect** (`list` of `int`) – List specifying the top-left and bottom-right coordinates of the region to copy from (`<X0>`, `<Y0>`, `<X1>`, `<Y1>`).
- **dstRect** (`list` of `int` or `None`) – List specifying the top-left and bottom-right coordinates of the region to copy to (`<X0>`, `<Y0>`, `<X1>`, `<Y1>`). If `None`, `srcRect` is used for `dstRect`.
- **filter** (`int`) – Interpolation method to use if the image is stretched, default is `GL_LINEAR`, but can also be `GL_NEAREST`.

Returns

Return type None

Examples

Blitting pixels from on FBO to another:

```
# bind framebuffer to read pixels from
GL.glBindFramebuffer(GL.GL_READ_FRAMEBUFFER, srcFbo)

# bind framebuffer to draw pixels to
GL.glBindFramebuffer(GL.GL_DRAW_FRAMEBUFFER, dstFbo)

gltools.blitFBO((0,0,800,600), (0,0,800,600))

# unbind both read and draw buffers
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, 0)
```

psychopy.tools.gltools.useFBO

`psychopy.tools.gltools.useFBO` (*fbo*)

Context manager for Framebuffer Object bindings. This function yields the framebuffer name as an integer.

:param `fbo` `int` or `Framebuffer`: OpenGL Framebuffer Object name/ID or descriptor.

Yields `int` – OpenGL name of the framebuffer bound in the context.

Examples

Using a framebuffer context manager:

```
# FBO bound somewhere deep in our code
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, someOtherFBO)

...

# create a new FBO, but we have no idea what the currently bound FBO is
fbo = createFBO()

# use a context to bind attachments
with bindFBO(fbo):
    attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
    attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
    attach(GL.GL_STENCIL_ATTACHMENT, depthRb)
    isComplete = gltools.isComplete()

# someOtherFBO is still bound!
```

Renderbuffers

Tools for creating Renderbuffers.

<code>createRenderbuffer</code> (width, height[, ...])	Create a new Renderbuffer Object with a specified internal format.
<code>deleteRenderbuffer</code> (renderBuffer)	Free the resources associated with a renderbuffer.

psychoPy.tools.gltools.createRenderbuffer

`psychoPy.tools.gltools.createRenderbuffer` (*width*, *height*, *internalFormat=32856*, *samples=1*)

Create a new Renderbuffer Object with a specified internal format. A multisample storage buffer is created if *samples* > 1.

Renderbuffers contain image data and are optimized for use as render targets. See https://www.khronos.org/opengl/wiki/Renderbuffer_Object for more information.

Parameters

- **width** (*int*) – Buffer width in pixels.
- **height** (*int*) – Buffer height in pixels.
- **internalFormat** (*int*) – Format for renderbuffer data (e.g. `GL_RGBA8`, `GL_DEPTH24_STENCIL8`).
- **samples** (*int*) – Number of samples for multi-sampling, should be >1 and power-of-two. Work with one sample, but will raise a warning.

Returns A descriptor of the created renderbuffer.

Return type Renderbuffer

Notes

The ‘*userData*’ field of the returned descriptor is a dictionary that can be used to store arbitrary data associated with the buffer.

psychoPy.tools.gltools.deleteRenderbuffer

`psychoPy.tools.gltools.deleteRenderbuffer` (*renderBuffer*)

Free the resources associated with a renderbuffer. This invalidates the renderbuffer’s ID.

Textures

Tools for creating textures.

<code>createTexImage2D</code> (width, height[, target, ...])	Create a 2D texture in video memory.
<code>createTexImage2dFromFile</code> (imgFile[, transpose])	Load an image from file directly into a texture.
<code>createTexImage2DMultisample</code> (width, height[, ...])	Create a 2D multisampled texture.

continues on next page

Table 9.57 – continued from previous page

<code>deleteTexture(texture)</code>	Free the resources associated with a texture.
<code>bindTexture(texture[, unit, enable])</code>	Bind a texture.
<code>unbindTexture([texture])</code>	Unbind a texture.
<code>createCubeMap(width, height[, target, ...])</code>	Create a cubemap.

psychopy.tools.gltools.createTexImage2D

`psychopy.tools.gltools.createTexImage2D` (*width*, *height*, *target=3553*, *level=0*, *internalFormat=32856*, *pixelFormat=6408*, *dataType=5126*, *data=None*, *unpackAlignment=4*, *texParams=None*)

Create a 2D texture in video memory. This can only create a single 2D texture with targets `GL_TEXTURE_2D` or `GL_TEXTURE_RECTANGLE`.

Parameters

- **width** (*int*) – Texture width in pixels.
- **height** (*int*) – Texture height in pixels.
- **target** (*int*) – The target texture should only be either `GL_TEXTURE_2D` or `GL_TEXTURE_RECTANGLE`.
- **level** (*int*) – LOD number of the texture, should be 0 if `GL_TEXTURE_RECTANGLE` is the target.
- **internalFormat** (*int*) – Internal format for texture data (e.g. `GL_RGBA8`, `GL_R11F_G11F_B10F`).
- **pixelFormat** (*int*) – Pixel data format (e.g. `GL_RGBA`, `GL_DEPTH_STENCIL`)
- **dataType** (*int*) – Data type for pixel data (e.g. `GL_FLOAT`, `GL_UNSIGNED_BYTE`).
- **data** (*ctypes* or *None*) – Ctypes pointer to image data. If *None* is specified, the texture will be created but pixel data will be uninitialized.
- **unpackAlignment** (*int*) – Alignment requirements of each row in memory. Default is 4.
- **texParams** (*dict*) – Optional texture parameters specified as *dict*. These values are passed to *glTexParameteri*. Each tuple must contain a parameter name and value. For example, `texParameters={GL.GL_TEXTURE_MIN_FILTER: GL.GL_LINEAR, GL.GL_TEXTURE_MAG_FILTER: GL.GL_LINEAR}`.

Returns A *TexImage2D* descriptor.

Return type *TexImage2D*

Notes

The 'userData' field of the returned descriptor is a dictionary that can be used to store arbitrary data associated with the texture.

Previous textures are unbound after calling 'createTexImage2D'.

Examples

Creating a texture from an image file:

```
import pyglet.gl as GL # using Pyglet for now

# empty texture
textureDesc = createTexImage2D(1024, 1024, internalFormat=GL.GL_RGBA8)

# load texture data from an image file using Pillow and NumPy
from PIL import Image
import numpy as np
im = Image.open(imageFile) # 8bpp!
im = im.transpose(Image.FLIP_TOP_BOTTOM) # OpenGL origin is at bottom
im = im.convert("RGBA")
pixelData = np.array(im).ctypes # convert to ctypes!

width = pixelData.shape[1]
height = pixelData.shape[0]
textureDesc = gltools.createTexImage2D(
    width,
    height,
    internalFormat=GL.GL_RGBA,
    pixelFormat=GL.GL_RGBA,
    dataType=GL.GL_UNSIGNED_BYTE,
    data=pixelData,
    unpackAlignment=1,
    texParameters=[(GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR),
                    (GL.GL_TEXTURE_MIN_FILTER, GL.GL_LINEAR)])

GL.glBindTexture(GL.GL_TEXTURE_2D, textureDesc.id)
```

psychoPy.tools.gltools.createTexImage2dFromFile

psychoPy.tools.gltools.**createTexImage2dFromFile** (*imgFile*, *transpose=True*)

Load an image from file directly into a texture.

This is a convenience function to quickly get an image file loaded into a 2D texture. The image is converted to RGBA format. Texture parameters are set for linear interpolation.

Parameters

- **imgFile** (*str*) – Path to the image file.
- **transpose** (*bool*) – Flip the image so it appears upright when displayed in OpenGL image coordinates.

Returns Texture descriptor.

Return type `TexImage2D`

psychopy.tools.gltools.createTexImage2DMultisample

`psychopy.tools.gltools.createTexImage2DMultisample` (*width*, *height*, *target=37120*, *samples=1*, *internalFormat=32856*, *texParameters=()*)

Create a 2D multisampled texture.

Parameters

- **width** (*int*) – Texture width in pixels.
- **height** (*int*) – Texture height in pixels.
- **target** (*int*) – The target texture (e.g. `GL_TEXTURE_2D_MULTISAMPLE`).
- **samples** (*int*) – Number of samples for multi-sampling, should be >1 and power-of-two. Work with one sample, but will raise a warning.
- **internalFormat** (*int*) – Internal format for texture data (e.g. `GL_RGBA8`, `GL_R11F_G11F_B10F`).
- **texParameters** (*list of tuple of int*) – Optional texture parameters specified as a list of tuples. These values are passed to ‘glTexParameter’. Each tuple must contain a parameter name and value. For example, `texParameters=[(GL.GL_TEXTURE_MIN_FILTER, GL.GL_LINEAR), (GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR)]`

Returns A `TexImage2DMultisample` descriptor.

Return type `TexImage2DMultisample`

psychopy.tools.gltools.deleteTexture

`psychopy.tools.gltools.deleteTexture` (*texture*)

Free the resources associated with a texture. This invalidates the texture’s ID.

psychopy.tools.gltools.bindTexture

`psychopy.tools.gltools.bindTexture` (*texture*, *unit=None*, *enable=True*)

Bind a texture.

Function binds *texture* to *unit* (if specified). If *unit* is `None`, the texture will be bound but not assigned to a texture unit.

Parameters

- **texture** (*TexImage2D*) – Texture descriptor to bind.
- **unit** (*int*, *optional*) – Texture unit to associated the texture with.
- **enable** (*bool*) – Enable textures upon binding.

psychopy.tools.gltools.unbindTexture

`psychopy.tools.gltools.unbindTexture` (*texture=None*)
 Unbind a texture.

Parameters `texture` (*TexImage2D*) – Texture descriptor to unbind.

psychopy.tools.gltools.createCubeMap

`psychopy.tools.gltools.createCubeMap` (*width, height, target=34067, level=0, internalFormat=6408, pixelFormat=6408, dataType=5121, data=None, unpackAlignment=4, texParams=None*)

Create a cubemap.

Parameters

- **name** (*int* or *GLuint*) – OpenGL handle for the cube map. Is 0 if uninitialized.
- **target** (*int*) – The target texture should only be `GL_TEXTURE_CUBE_MAP`.
- **width** (*int*) – Texture width in pixels.
- **height** (*int*) – Texture height in pixels.
- **level** (*int*) – LOD number of the texture.
- **internalFormat** (*int*) – Internal format for texture data (e.g. `GL_RGBA8`, `GL_R11F_G11F_B10F`).
- **pixelFormat** (*int*) – Pixel data format (e.g. `GL_RGBA`, `GL_DEPTH_STENCIL`)
- **dataType** (*int*) – Data type for pixel data (e.g. `GL_FLOAT`, `GL_UNSIGNED_BYTE`).
- **data** (*list* or *tuple*) – List of six ctypes pointers to image data for each cubemap face. Image data is assigned to a face by index [+X, -X, +Y, -Y, +Z, -Z]. All images must have the same size as specified by *width* and *height*.
- **unpackAlignment** (*int*) – Alignment requirements of each row in memory. Default is 4.
- **texParams** (*list* of *tuple* of *int*) – Optional texture parameters specified as *dict*. These values are passed to `glTexParameteri`. Each tuple must contain a parameter name and value. For example, `texParameters={ GL_TEXTURE_MIN_FILTER: GL_LINEAR, GL_TEXTURE_MAG_FILTER: GL_LINEAR}`. These can be changed and will be updated the next time this instance is passed to `bindTexture()`.

Vertex Buffer/Array Objects

Tools for creating and working with Vertex Buffer Objects (VBOs) and Vertex Array Objects (VAOs).

<code>VertexArrayInfo</code> ([name, count, ...])	Vertex array object (VAO) descriptor.
<code>createVAO</code> (attribBuffers[, indexBuffer, ...])	Create a Vertex Array object (VAO).
<code>drawVAO</code> (vao[, mode, start, count, ...])	Draw a vertex array object.
<code>deleteVAO</code> (vao)	Delete a Vertex Array Object (VAO).
<code>VertexBufferInfo</code> ([name, target, usage, ...])	Vertex buffer object (VBO) descriptor.
<code>createVBO</code> (data[, target, dataType, usage])	Create an array buffer object (VBO).
<code>bindVBO</code> (vbo)	Bind a VBO to the current GL state.

continues on next page

Table 9.58 – continued from previous page

<code>unbindVBO(vbo)</code>	Unbind a vertex buffer object (VBO).
<code>mapBuffer(vbo[, start, length, read, write, ...])</code>	Map a vertex buffer object to client memory.
<code>unmapBuffer(vbo)</code>	Unmap a previously mapped buffer.
<code>deleteVBO(vbo)</code>	Delete a vertex buffer object (VBO).
<code>setVertexAttribPointer(index, vbo[, size, ...])</code>	Define an array of vertex attribute data with a VBO descriptor.
<code>enableVertexAttribArray(index[, legacy])</code>	Enable a vertex attribute array.
<code>disableVertexAttribArray(index[, legacy])</code>	Disable a vertex attribute array.

psychopy.tools.gltools.VertexArrayInfo

```
class psychopy.tools.gltools.VertexArrayInfo (name=0, count=0, activeAttribs=None,
                                              indexBuffer=None, attribDivisors=None,
                                              isLegacy=False, userData=None)
```

Vertex array object (VAO) descriptor.

This class only stores information about the VAO it refers to, it does not contain any actual array data associated with the VAO. Calling `createVAO()` returns instances of this class.

If `isLegacy` is `True`, attribute binding states are using deprecated (but still supported) pointer definition calls (eg. `glVertexPointer`). This is to ensure backwards compatibility. The keys stored in `activeAttribs` must be `GLenum` types such as `GL_VERTEX_ARRAY`.

Parameters

- **name** (*int*) – OpenGL handle for the VAO.
- **count** (*int*) – Number of vertex elements. If `indexBuffer` is not `None`, count corresponds to the number of elements in the index buffer.
- **activeAttribs** (*dict*) – Attributes and buffers defined as part of this VAO state. Keys are attribute pointer indices or capabilities (ie. `GL_VERTEX_ARRAY`). Modifying these values will not update the VAO state.
- **indexBuffer** (*VertexBufferInfo, optional*) – Buffer object for indices.
- **attribDivisors** (*dict, optional*) – Divisors for each attribute.
- **isLegacy** (*bool*) – Array pointers were defined using the deprecated OpenGL API. If `True`, the VAO may work with older GLSL shaders versions and the fixed-function pipeline.
- **userData** (*dict or None, optional*) – Optional user defined data associated with this VAO.

```
__init__ (name=0, count=0, activeAttribs=None, indexBuffer=None, attribDivisors=None, isLegacy=False, userData=None)
```

Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__</code> ([name, count, activeAttribs, ...])	Initialize self.
---	------------------

Attributes

<code>activeAttribs</code>
<code>attribDivisors</code>
<code>count</code>
<code>indexBuffer</code>
<code>isLegacy</code>
<code>name</code>
<code>userData</code>

psychoPy.tools.gltools.createVAO

`psychoPy.tools.gltools.createVAO` (*attribBuffers*, *indexBuffer=None*, *attribDivisors=None*, *legacy=False*)

Create a Vertex Array object (VAO). VAOs store buffer binding states, reducing CPU overhead when drawing objects with vertex data stored in VBOs.

Define vertex attributes within a VAO state by passing a mapping for generic attribute indices and VBO buffers.

Parameters

- **attribBuffers** (*dict*) – Attributes and associated VBOs to add to the VAO state. Keys are vertex attribute pointer indices, values are VBO descriptors to define. Values can be *tuples* where the first value is the buffer descriptor, the second is the number of attribute components (*int*, either 2, 3 or 4), the third is the offset (*int*), and the last is whether to normalize the array (*bool*).
- **indexBuffer** (*VertexBufferInfo*) – Optional index buffer.
- **attribDivisors** (*dict*) – Attribute divisors to set. Keys are vertex attribute pointer indices, values are the number of instances that will pass between updates of an attribute. Setting attribute divisors is only permitted if *legacy* is *False*.
- **legacy** (*bool*, *optional*) – Use legacy attribute pointer functions when setting the VAO state. This is for compatibility with older GL implementations. Key specified to *attribBuffers* must be *GLenum* types such as *GL_VERTEX_ARRAY* to indicate the capability to use.

Examples

Create a vertex array object and enable buffer states within it:

```
vao = createVAO({0: vertexPos, 1: texCoords, 2: vertexNormals})
```

Using an interleaved vertex buffer, all attributes are in the same buffer (*vertexAttr*). We need to specify offsets for each attribute by passing a buffer in a *tuple* with the second value specifying the offset:

```
# buffer with interleaved layout `00011222` per-attribute
vao = createVAO(
    {0: (vertexAttr, 3),           # size 3, offset 0
     1: (vertexAttr, 2, 3),       # size 2, offset 3
     2: (vertexAttr, 3, 5, True)} # size 3, offset 5, normalize
```

You can mix interleaved and single-use buffers:

```
vao = createVAO(
    {0: (vertexAttr, 3, 0), 1: (vertexAttr, 3, 3), 2: vertexColors})
```

Specifying an optional index array, this is used for indexed drawing of primitives:

```
vao = createVAO({0: vertexPos}, indexBuffer=indices)
```

The returned *VertexArrayInfo* instance will have attribute `isIndexed==True`.

Drawing vertex arrays using a VAO, will use the *indexBuffer* if available:

```
# draw the array
drawVAO(vao, mode=GL.GL_TRIANGLES)
```

Use legacy attribute pointer bindings when building a VAO for compatibility with the fixed-function pipeline and older GLSL versions:

```
attribBuffers = {GL_VERTEX_ARRAY: vertexPos, GL_NORMAL_ARRAY: normals}
vao = createVAO(attribBuffers, legacy=True)
```

If you wish to used instanced drawing, you can specify attribute divisors this way:

```
vao = createVAO(
    {0: (vertexAttr, 3, 0), 1: (vertexAttr, 3, 3), 2: vertexColors},
    attribDivisors={2: 1})
```

psychopy.tools.glttools.drawVAO

`psychopy.tools.glttools.drawVAO(vao, mode=4, start=0, count=None, instanceCount=None, flush=False)`

Draw a vertex array object. Uses *glDrawArrays* or *glDrawElements* if *instanceCount* is *None*, or else *glDrawArraysInstanced* or *glDrawElementsInstanced* is used.

Parameters

- **vao** (*VertexArrayObject*) – Vertex Array Object (VAO) to draw.
- **mode** (*int, optional*) – Drawing mode to use (e.g. `GL_TRIANGLES`, `GL_QUADS`, `GL_POINTS`, etc.)
- **start** (*int, optional*) – Starting index for array elements. Default is `0` which is the beginning of the array.
- **count** (*int, optional*) – Number of indices to draw from *start*. Must not exceed `vao.count - start`.
- **instanceCount** (*int or None*) – Number of instances to draw. If `>0` and not *None*, instanced drawing will be used.
- **flush** (*bool, optional*) – Flush queued drawing commands before returning.

Examples

Creating a VAO and drawing it:

```
# draw the VAO, renders the mesh
drawVAO(vaoDesc, GL.GL_TRIANGLES)
```

psychopy.tools.gltools.deleteVAO

`psychopy.tools.gltools.deleteVAO(vao)`

Delete a Vertex Array Object (VAO). This does not delete array buffers bound to the VAO.

Parameters `vao` (`VertexArrayInfo`) – VAO to delete. All fields in the descriptor except `userData` will be reset.

psychopy.tools.gltools.VertexBufferInfo

`class psychopy.tools.gltools.VertexBufferInfo(name=0, target=34962, usage=35044, dataType=5126, size=0, stride=0, shape=0, userData=None)`

Vertex buffer object (VBO) descriptor.

This class only stores information about the VBO it refers to, it does not contain any actual array data associated with the VBO. Calling `createVBO()` returns instances of this class.

It is recommended to use `gltools` functions `bindVBO()`, `unbindVBO()`, `mapBuffer()`, etc. when working with these objects.

Parameters

- **name** (`GLuint` or `int`) – OpenGL handle for the buffer.
- **target** (`GLenum` or `int`, *optional*) – Target used when binding the buffer (e.g. `GL_VERTEX_ARRAY` or `GL_ELEMENT_ARRAY_BUFFER`). Default is `GL_VERTEX_ARRAY`
- **usage** (`GLenum` or `int`, *optional*) – Usage type for the array (i.e. `GL_STATIC_DRAW`).
- **dataType** (`GLenum`, *optional*) – Data type of array. Default is `GL_FLOAT`.
- **size** (`int`, *optional*) – Size of the buffer in bytes.
- **stride** (`int`, *optional*) – Number of bytes between adjacent attributes. If 0, values are assumed to be tightly packed.
- **shape** (`tuple` or `list`, *optional*) – Shape of the array used to create this VBO.
- **userData** (`dict`, *optional*) – Optional user defined data associated with the VBO. If `None`, `userData` will be initialized as an empty dictionary.

`__init__` (`name=0, target=34962, usage=35044, dataType=5126, size=0, stride=0, shape=0, userData=None`)

Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__</code> (name, target, usage, dataType, ...)	Initialize self.
<code>validate</code> ()	Check if the data contained in this descriptor matches what is actually present in the OpenGL state.

Attributes

<code>dataType</code>	
<code>hasBuffer</code>	Check if the VBO assigned to <i>name</i> is a buffer.
<code>isIndex</code>	<i>True</i> if the buffer referred to by this object is an index array.
<code>name</code>	
<code>shape</code>	
<code>size</code>	
<code>stride</code>	
<code>target</code>	
<code>usage</code>	
<code>userData</code>	

psychopy.tools.gltools.createVBO

`psychopy.tools.gltools.createVBO` (*data*, *target*=34962, *dataType*=5126, *usage*=35044)

Create an array buffer object (VBO).

Creates a VBO using input data, usually as a *ndarray* or *list*. Attributes common to one vertex should occupy a single row of the *data* array.

Parameters

- **data** (*array_like*) – A 2D array of values to write to the array buffer. The data type of the VBO is inferred by the type of the array. If the input is a Python *list* or *tuple* type, the data type of the array will be *GL_FLOAT*.
- **target** (*int*) – Target used when binding the buffer (e.g. *GL_VERTEX_ARRAY* or *GL_ELEMENT_ARRAY_BUFFER*). Default is *GL_VERTEX_ARRAY*.
- **dataType** (*Glenum*, *optional*) – Data type of array. Input data will be recast to an appropriate type if necessary. Default is *GL_FLOAT*.
- **usage** (*Glenum* or *int*, *optional*) – Usage type for the array (i.e. *GL_STATIC_DRAW*).

Returns A descriptor with vertex buffer information.

Return type *VertexBufferInfo*

Examples

Creating a vertex buffer object with vertex data:

```
# vertices of a triangle
verts = [[ 1.0,  1.0, 0.0], # v0
         [ 0.0, -1.0, 0.0], # v1
         [-1.0,  1.0, 0.0]] # v2

# load vertices to graphics device, return a descriptor
vboDesc = createVBO(verts)
```

Drawing triangles or quads using vertex buffer data:

```
nIndices, vSize = vboDesc.shape # element size

bindVBO(vboDesc)
setVertexAttribPointer(
    GL_VERTEX_ARRAY, vSize, vboDesc.dataType, legacy=True)
enableVertexAttribArray(GL_VERTEX_ARRAY, legacy=True)

if vSize == 3:
    drawMode = GL_TRIANGLES
elif vSize == 4:
    drawMode = GL_QUADS

glDrawArrays(drawMode, 0, nIndices)
glFlush()

disableVertexAttribArray(GL_VERTEX_ARRAY, legacy=True)
unbindVBO()
```

Custom data can be associated with this vertex buffer by specifying *userData*:

```
myVBO = createVBO(data)
myVBO.userData['startIdx'] = 14 # first index to draw with

# use it later
nIndices, vSize = vboDesc.shape # element size
startIdx = myVBO.userData['startIdx']
endIdx = nIndices - startIdx
glDrawArrays(GL_TRIANGLES, startIdx, endIdx)
glFlush()
```

psychopy.tools.gltools.bindVBO

`psychopy.tools.gltools.bindVBO(vbo)`

Bind a VBO to the current GL state.

Parameters `vbo` (`VertexBufferInfo`) – VBO descriptor to bind.

Returns `True` is the binding state was changed. Returns `False` if the state was not changed due to the buffer already being bound.

Return type `bool`

psychopy.tools.gltools.unbindVBO

`psychopy.tools.gltools.unbindVBO(vbo)`

Unbind a vertex buffer object (VBO).

Parameters `vbo` (`VertexBufferInfo`) – VBO descriptor to unbind.

psychopy.tools.gltools.mapBuffer

`psychopy.tools.gltools.mapBuffer(vbo, start=0, length=None, read=True, write=True, noSync=False)`

Map a vertex buffer object to client memory. This allows you to modify its contents.

If planning to update VBO vertex data, make sure the VBO *usage* types are `GL_DYNAMIC_*` or `GL_STREAM_*` or else serious performance issues may arise.

Warning: Modifying buffer data must be done carefully, or else system stability may be affected. Do not use the returned view `ndarray` outside of successive `mapBuffer()` and `unmapBuffer()` calls. Do not use the mapped buffer for rendering until after `unmapBuffer()` is called.

Parameters

- **vbo** (`VertexBufferInfo`) – Vertex buffer to map to client memory.
- **start** (`int`) – Initial index of the sub-range of the buffer to modify.
- **length** (`int` or `None`) – Number of elements of the sub-array to map from *offset*. If `None`, all elements to from *offset* to the end of the array are mapped.
- **read** (`bool`, *optional*) – Allow data to be read from the buffer (sets `GL_MAP_READ_BIT`). This is ignored if `noSync` is `True`.
- **write** (`bool`, *optional*) – Allow data to be written to the buffer (sets `GL_MAP_WRITE_BIT`).
- **noSync** (`bool`, *optional*) – If `True`, GL will not wait until the buffer is free (i.e. not being processed by the GPU) to map it (sets `GL_MAP_UNSYNCHRONIZED_BIT`). The contents of the previous storage buffer are discarded and the driver returns a new one. This prevents the CPU from stalling until the buffer is available.

Returns View of the data. The type of the returned array is one which best matches the data type of the buffer.

Return type `ndarray`

Examples

Map a buffer and edit it:

```
arr = mapBuffer(vbo)
arr[:, :] += 2.0 # add 2 to all values
unmapBuffer(vbo) # call when done
# Don't ever modify `arr` after calling `unmapBuffer`. Delete it if
# necessary to prevent it from being used.
del arr
```

Modify a sub-range of data by specifying *start* and *length*, indices correspond to values, not byte offsets:

```
arr = mapBuffer(vbo, start=12, end=24)
arr[:, :] *= 10.0
unmapBuffer(vbo)
```

psychoPy.tools.gltools.unmapBuffer

`psychoPy.tools.gltools.unmapBuffer(vbo)`

Unmap a previously mapped buffer. Must be called after `mapBuffer()` is called and before any drawing operations which use the buffer are called. Failing to call this before using the buffer could result in a system error.

Parameters `vbo` (`VertexBufferInfo`) – Vertex buffer descriptor.

Returns `True` if the buffer has been successfully modified. If `False`, the data was corrupted for some reason and needs to be resubmitted.

Return type `bool`

psychoPy.tools.gltools.deleteVBO

`psychoPy.tools.gltools.deleteVBO(vbo)`

Delete a vertex buffer object (VBO).

Parameters `vbo` (`VertexBufferInfo`) – Descriptor of VBO to delete.

psychoPy.tools.gltools.setVertexAttribPointer

`psychoPy.tools.gltools.setVertexAttribPointer(index, vbo, size=None, offset=0, normalize=False, legacy=False)`

Define an array of vertex attribute data with a VBO descriptor.

In modern OpenGL implementations, attributes are ‘generic’, where an attribute pointer index does not correspond to any special vertex property. Usually the usage for an attribute is defined in the shader program. It is recommended that shader programs define attributes using the *layout* parameters:

```
layout (location = 0) in vec3 position;
layout (location = 1) in vec2 texCoord;
layout (location = 2) in vec3 normal;
```

Setting attribute pointers can be done like this:

```
setVertexAttribPointer(0, posVbo)
setVertexAttribPointer(1, texVbo)
setVertexAttribPointer(2, normVbo)
```

For compatibility with older OpenGL specifications, some drivers will alias vertex pointers unless they are explicitly defined in the shader. This allows VAOs to be used with the fixed-function pipeline or older GLSL versions.

On nVidia graphics drivers (and maybe others), the following attribute pointers indices are aliased with reserved GLSL names:

- `gl_Vertex - 0`

- `gl_Normal` - 2
- `gl_Color` - 3
- `gl_SecondaryColor` - 4
- `gl_FogCoord` - 5
- `gl_MultiTexCoord0` - 8
- `gl_MultiTexCoord1` - 9
- `gl_MultiTexCoord2` - 10
- `gl_MultiTexCoord3` - 11
- `gl_MultiTexCoord4` - 12
- `gl_MultiTexCoord5` - 13
- `gl_MultiTexCoord6` - 14
- `gl_MultiTexCoord7` - 15

Specifying *legacy* as *True* will allow for old-style pointer definitions. You must specify the capability as a *GLenum* associated with the pointer in this case:

```
setVertexAttribPointer(GL_VERTEX_ARRAY, posVbo, legacy=True)
setVertexAttribPointer(GL_TEXTURE_COORD_ARRAY, texVbo, legacy=True)
setVertexAttribPointer(GL_NORMAL_ARRAY, normVbo, legacy=True)
```

Parameters

- **index** (*int*) – Index of the attribute to modify. If *legacy=True*, this value should be a *GLenum* type corresponding to the capability to bind the buffer to, such as *GL_VERTEX_ARRAY*, *GL_TEXTURE_COORD_ARRAY*, *GL_NORMAL_ARRAY*, etc.
- **vbo** (*VertexBufferInfo*) – VBO descriptor.
- **size** (*int, optional*) – Number of components per vertex attribute, can be either 1, 2, 3, or 4. If *None* is specified, the component size will be inferred from the *shape* of the VBO. You must specify this value if the VBO is interleaved.
- **offset** (*int, optional*) – Starting index of the attribute in the buffer.
- **normalize** (*bool, optional*) – Normalize fixed-point format values when accessed.
- **legacy** (*bool, optional*) – Use legacy vertex attributes (ie. *GL_VERTEX_ARRAY*, *GL_TEXTURE_COORD_ARRAY*, etc.) for backwards compatibility.

Examples

Define a generic attribute from a vertex buffer descriptor:

```
# set the vertex location attribute
setVertexAttribPointer(0, vboDesc) # 0 is vertex in our shader
GL.glColor3f(1.0, 0.0, 0.0) # red triangle

# draw the triangle
nIndices, vSize = vboDesc.shape # element size
GL.glDrawArrays(GL.GL_TRIANGLES, 0, nIndices)
```

If our VBO has interleaved attributes, we can specify *offset* to account for that:

```

# define interleaved vertex attributes
#      /      Position      / Texture / Normals      /
vQuad = [[ -1.0, -1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0], # v0
          [ -1.0,  1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0], # v1
          [  1.0,  1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0], # v2
          [  1.0, -1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0]] # v3

# create a VBO with interleaved attributes
vboInterleaved = createVBO(np.asarray(vQuad, dtype=np.float32))

# ... before rendering, set the attribute pointers
GL.glBindBuffer(vboInterleaved.target, vboInterleaved.name)
gltools.setVertexAttribPointer(
    0, vboInterleaved, size=3, offset=0) # vertex pointer
gltools.setVertexAttribPointer(
    8, vboInterleaved, size=2, offset=3) # texture pointer
gltools.setVertexAttribPointer(
    3, vboInterleaved, size=3, offset=5) # normals pointer

# Note, we specified `bind=False` since we are managing the binding
# state. It is recommended that you do this when setting up interleaved
# buffers to avoid re-binding the same buffer.

# draw red, full screen quad
GL.glColor3f(1.0, 0.0, 0.0)
GL.glDrawArrays(GL.GL_QUADS, 0, vboInterleaved.shape[1])

# call these when done if `enable=True`
gltools.disableVertexAttribArray(0)
gltools.disableVertexAttribArray(8)
gltools.disableVertexAttribArray(1)

# unbind the buffer
GL.glBindBuffer(vboInterleaved.target, 0)

```

psychoPy.tools.gltools.enableVertexAttribArray

psychoPy.tools.gltools.**enableVertexAttribArray** (*index*, *legacy=False*)

Enable a vertex attribute array. Attributes will be used for use by subsequent draw operations. Be sure to call `disableVertexAttribArray()` on the same attribute to prevent currently enabled attributes from affecting later rendering.

Parameters

- **index** (*int*) – Index of the attribute to enable. If *legacy=True*, this value should be a *GLenum* type corresponding to the capability to bind the buffer to, such as `GL_VERTEX_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, `GL_NORMAL_ARRAY`, etc.
- **legacy** (*bool*, *optional*) – Use legacy vertex attributes (ie. `GL_VERTEX_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, etc.) for backwards compatibility.

psychopy.tools.gltools.disableVertexAttribArray

`psychopy.tools.gltools.disableVertexAttribArray` (*index*, *legacy=False*)

Disable a vertex attribute array.

Parameters

- **index** (*int*) – Index of the attribute to enable. If *legacy=True*, this value should be a *GLenum* type corresponding to the capability to bind the buffer to, such as *GL_VERTEX_ARRAY*, *GL_TEXTURE_COORD_ARRAY*, *GL_NORMAL_ARRAY*, etc.
- **legacy** (*bool*, *optional*) – Use legacy vertex attributes (ie. *GL_VERTEX_ARRAY*, *GL_TEXTURE_COORD_ARRAY*, etc.) for backwards compatibility.

Materials and Lighting

Tools for specifying the appearance of faces and shading. Note that these tools use the legacy OpenGL pipeline which may not be available on your platform. Use fragment/vertex shaders instead for newer applications.

<code>createMaterial</code> ([params, textures, face])	Create a new material.
<code>useMaterial</code> (material[, useTextures])	Use a material for proceeding vertex draws.
<code>createLight</code> ([params])	Create a point light source.
<code>useLights</code> (lights[, setupOnly])	Use specified lights in successive rendering operations.
<code>setAmbientLight</code> (color)	Set the global ambient lighting for the scene when lighting is enabled.

psychopy.tools.gltools.createMaterial

`psychopy.tools.gltools.createMaterial` (*params=()*, *textures=()*, *face=1032*)

Create a new material.

Parameters

- **params** (*list of tuple*, *optional*) – List of material modes and values. Each mode is assigned a value as (mode, color). Modes can be *GL_AMBIENT*, *GL_DIFFUSE*, *GL_SPECULAR*, *GL_EMISSION*, *GL_SHININESS* or *GL_AMBIENT_AND_DIFFUSE*. Colors must be a tuple of 4 floats which specify reflectance values for each RGBA component. The value of *GL_SHININESS* should be a single float. If no values are specified, an empty material will be created.
- **textures** (*list of tuple*, *optional*) – List of texture units and *TexImage2D* descriptors. These will be written to the ‘textures’ field of the returned descriptor. For example, [(*GL_TEXTURE0*, *texDesc0*), (*GL_TEXTURE1*, *texDesc1*)]. The number of texture units per-material is *GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS*.
- **face** (*int*, *optional*) – Faces to apply material to. Values can be *GL_FRONT_AND_BACK*, *GL_FRONT* and *GL_BACK*. The default is *GL_FRONT_AND_BACK*.

Returns A descriptor with material properties.

Return type Material

Examples

Creating a new material with given properties:

```
# The values for the material below can be found at
# http://devernay.free.fr/cours/opengl/materials.html

# create a gold material
gold = createMaterial([
    (GL.GL_AMBIENT, (0.24725, 0.19950, 0.07450, 1.0)),
    (GL.GL_DIFFUSE, (0.75164, 0.60648, 0.22648, 1.0)),
    (GL.GL_SPECULAR, (0.628281, 0.555802, 0.366065, 1.0)),
    (GL.GL_SHININESS, 0.4 * 128.0)])
```

Use the material when drawing:

```
useMaterial(gold)
drawVAO( ... ) # all meshes will be gold
useMaterial(None) # turn off material when done
```

Create a red plastic material, but define reflectance and shine later:

```
red_plastic = createMaterial()

# you need to convert values to ctypes!
red_plastic.values[GL_AMBIENT] = (GLfloat * 4)(0.0, 0.0, 0.0, 1.0)
red_plastic.values[GL_DIFFUSE] = (GLfloat * 4)(0.5, 0.0, 0.0, 1.0)
red_plastic.values[GL_SPECULAR] = (GLfloat * 4)(0.7, 0.6, 0.6, 1.0)
red_plastic.values[GL_SHININESS] = 0.25 * 128.0

# set and draw
useMaterial(red_plastic)
drawVertexbuffers( ... ) # all meshes will be red plastic
useMaterial(None)
```

psychoPy.tools.gltools.useMaterial

`psychoPy.tools.gltools.useMaterial(material, useTextures=True)`

Use a material for proceeding vertex draws.

Parameters

- **material** (Material or None) – Material descriptor to use. Default material properties are set if None is specified. This is equivalent to disabling materials.
- **useTextures** (bool) – Enable textures. Textures specified in a material descriptor’s ‘texture’ attribute will be bound and their respective texture units will be enabled. Note, when disabling materials, the value of useTextures must match the previous call. If there are no textures attached to the material, useTexture will be silently ignored.

Returns

Return type None

Notes

1. If a material mode has a value of None, a color with all components 0.0 will be assigned.
2. Material colors and shininess values are accessible from shader programs after calling ‘useMaterial’. Values can be accessed via built-in ‘gl_FrontMaterial’ and ‘gl_BackMaterial’ structures (e.g. gl_FrontMaterial.diffuse).

Examples

Use a material when drawing:

```
useMaterial(metalMaterials.gold)
drawVAO( ... ) # all meshes drawn will be gold
useMaterial(None) # turn off material when done
```

psychopy.tools.gltools.createLight

`psychopy.tools.gltools.createLight` (*params=()*)
Create a point light source.

psychopy.tools.gltools.useLights

`psychopy.tools.gltools.useLights` (*lights, setupOnly=False*)
Use specified lights in successive rendering operations. All lights will be transformed using the present modelview matrix.

Parameters

- **lights** (List of `Light` or `None`) – Descriptor of a light source. If `None`, lighting is disabled.
- **setupOnly** (`bool`, optional) – Do not enable lighting or lights. Specify `True` if lighting is being computed via fragment shaders.

psychopy.tools.gltools.setAmbientLight

`psychopy.tools.gltools.setAmbientLight` (*color*)
Set the global ambient lighting for the scene when lighting is enabled. This is equivalent to `GL.glLightModelfv(GL.GL_LIGHT_MODEL_AMBIENT, color)` and does not contribute to the `GL_MAX_LIGHTS` limit.

Parameters **color** (*tuple*) – Ambient lighting RGBA intensity for the whole scene.

Notes

If unset, the default value is (0.2, 0.2, 0.2, 1.0) when GL_LIGHTING is enabled.

Meshes

Tools for loading or procedurally generating meshes (3D models).

<code>ObjMeshInfo</code> ([vertexPos, texCoords, normals, ...])	Descriptor for mesh data loaded from a Wavefront OBJ file.
<code>loadObjFile</code> (objFile)	Load a Wavefront OBJ file (*.obj).
<code>loadMtlFile</code> (mtllib[, texParams])	Load a material library file (*.mtl).
<code>createUVSphere</code> ([radius, sectors, stacks, ...])	Create a UV sphere.
<code>createPlane</code> ([size])	Create a plane.
<code>createMeshGridFromArray</code> s(xvals, yvals[, ...])	Create a mesh grid using coordinates from arrays.
<code>createMeshGrid</code> ([size, subdiv, tessMode])	Create a grid mesh.
<code>createBox</code> ([size, flipFaces])	Create a box mesh.
<code>transformMeshPosOri</code> (vertices, normals[, ...])	Transform a mesh.
<code>calculateVertexNormals</code> (vertices, faces[, ...])	Calculate vertex normals given vertices and triangle faces.

psychopy.tools.gltools.ObjMeshInfo

```
class psychopy.tools.gltools.ObjMeshInfo (vertexPos=None, texCoords=None, normals=None, faces=None, extents=None, mtlFile=None)
```

Descriptor for mesh data loaded from a Wavefront OBJ file.

```
__init__ (vertexPos=None, texCoords=None, normals=None, faces=None, extents=None, mtlFile=None)
Initialize self. See help(type(self)) for accurate signature.
```

Methods

<code>__init__</code> ([vertexPos, texCoords, normals, ...])	Initialize self.
--	------------------

Attributes

<code>extents</code>
<code>faces</code>
<code>mtlFile</code>
<code>normals</code>
<code>texCoords</code>
<code>vertexPos</code>

psychopy.tools.gltools.loadObjFile

`psychopy.tools.gltools.loadObjFile(objFile)`

Load a Wavefront OBJ file (*.obj).

Loads vertex, normals, and texture coordinates from the provided *.obj file into arrays. These arrays can be processed then loaded into vertex buffer objects (VBOs) for rendering. The *.obj file must at least specify vertex position data to be loaded successfully. Normals and texture coordinates are optional.

Faces can be either triangles or quads, but not both. Faces are grouped by their materials. Index arrays are generated for each material present in the file.

Data from the returned *ObjMeshInfo* object can be used to create vertex buffer objects and arrays for rendering. See *Examples* below for details on how to do this.

Parameters `objFile` (`str`) – Path to the *.OBJ file to load.

Returns Mesh data.

Return type *ObjMeshInfo*

See also:

[`loadMtlFile\(\)`](#) Load a *.mtl file.

Notes

1. This importer should work fine for most sanely generated files. Export your model with Blender for best results, even if you used some other package to create it.
2. The mesh cannot contain both triangles and quads.

Examples

Loading a *.obj mode from file:

```
objModel = loadObjFile('/path/to/file.obj')
# load the material (*.mtl) file, textures are also loaded
mtllib = loadMtl('/path/to/' + objModel.mtlFile)
```

Creating separate vertex buffer objects (VBOs) for each vertex attribute:

```
vertexPosVBO = createVBO(objModel.vertexPos)
texCoordVBO = createVBO(objModel.texCoords)
normalsVBO = createVBO(objModel.normals)
```

Create vertex array objects (VAOs) to draw the mesh. We create VAOs for each face material:

```
objVAOs = {} # dictionary for VAOs
# for each material create a VAO
# keys are material names, values are index buffers
for material, faces in objModel.faces.items():
    # convert index buffer to VAO
    indexBuffer = gltools.createVBO(
        faces.flatten(), # flatten face index for element array
        target=GL.GL_ELEMENT_ARRAY_BUFFER,
        dataType=GL.GL_UNSIGNED_INT)
```

(continues on next page)

(continued from previous page)

```

# see `setVertexAttribPointer` for more information about attribute
# pointer indices
objVAOs[material] = gltools.createVAO(
    {0: vertexPosVBO, # 0 = gl_Vertex
     8: texCoordVBO, # 8 = gl_MultiTexCoord0
     2: normalsVBO}, # 2 = gl_Normal
    indexBuffer=indexBuffer)

# if using legacy attribute pointers, do this instead ...
# objVAOs[key] = createVAO({GL_VERTEX_ARRAY: vertexPosVBO,
#                             GL_TEXTURE_COORD_ARRAY: texCoordVBO,
#                             GL_NORMAL_ARRAY: normalsVBO},
#                             indexBuffer=indexBuffer,
#                             legacy=True) # this needs to be `True`

```

To render the VAOs using *objVAOs* created above, do the following:

```

for material, vao in objVAOs.items():
    useMaterial(mtlLib[material])
    drawVAO(vao)

useMaterial(None) # disable materials when done

```

Optionally, you can create a single-storage, interleaved VBO by using *numpy.hstack*. On some GL implementations, using single-storage buffers offers better performance:

```

interleavedData = numpy.hstack(
    (objModel.vertexPos, objModel.texCoords, objModel.normals))
vertexData = createVBO(interleavedData)

```

Creating VAOs with interleaved, single-storage buffers require specifying additional information, such as *size* and *offset*:

```

objVAOs = {}
for key, val in objModel.faces.items():
    indexBuffer = gltools.createVBO(
        faces.flatten(),
        target=GL.GL_ELEMENT_ARRAY_BUFFER,
        dataType=GL.GL_UNSIGNED_INT)

    objVAOs[key] = createVAO({0: (vertexData, 3, 0), # size=3, offset=0
                              8: (vertexData, 2, 3), # size=2, offset=3
                              2: (vertexData, 3, 5), # size=3, offset=5
                              indexBuffer=val})

```

Drawing VAOs with interleaved buffers is exactly the same as shown before with separate buffers.

psychopy.tools.gltools.loadMtlFile

psychopy.tools.gltools.**loadMtlFile** (*mtllib*, *texParams=None*)

Load a material library file (*.mtl).

Parameters

- **mtllib** (*str*) – Path to the material library file.
- **texParams** (*list or tuple*) – Optional texture parameters for loaded textures. Texture parameters are specified as a list of tuples. Each item specifies the option and parameter. For instance, `[(GL.GL_TEXTURE_MAG_FILTER, GL.GL_LINEAR), ...]`. By default, linear filtering is used for both the minifying and magnification filter functions. This is adequate for most uses.

Returns Dictionary of materials. Where each key is the material name found in the file, and values are *Material* namedtuple objects.

Return type dict

See also:

[loadObjFile\(\)](#) Load an *.OBJ file.

Examples

Load material associated with an *.OBJ file:

```
objModel = loadObjFile('/path/to/file.obj')
# load the material (*.mtl) file, textures are also loaded
mtllib = loadMtl('/path/to/' + objModel.mtlFile)
```

Use a material when rendering vertex arrays:

```
useMaterial(mtllib[material])
drawVAO(vao)
useMaterial(None) # disable materials when done
```

psychopy.tools.gltools.createUVSphere

psychopy.tools.gltools.**createUVSphere** (*radius=0.5*, *sectors=16*, *stacks=16*, *flipFaces=False*)

Create a UV sphere.

Procedurally generate a UV sphere by specifying its radius, and number of stacks and sectors. The poles of the resulting sphere will be aligned with the Z-axis.

Surface normals and texture coordinates are automatically generated. The returned normals are computed to produce smooth shading.

Parameters

- **radius** (*float, optional*) – Radius of the sphere in scene units (usually meters). Default is 0.5.
- **sectors** (*int, optional*) – Number of longitudinal and latitudinal sub-divisions. Default is 16 for both.

- **stacks** (*int, optional*) – Number of longitudinal and latitudinal sub-divisions. Default is 16 for both.
- **flipFaces** (*bool, optional*) – If *True*, normals and face windings will be set to point inward towards the center of the sphere. Texture coordinates will remain the same. Default is *False*.

Returns Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

Return type `tuple`

Examples

Create a UV sphere and VAO to render it:

```
vertices, textureCoords, normals, faces =          gltools.
↳createUVSphere(sectors=32, stacks=32)

vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(
    faces.flatten(),
    target=GL.GL_ELEMENT_ARRAY_BUFFER,
    dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO({0: vertexVBO, 8: texCoordVBO, 2: normalsVBO},
    indexBuffer=indexBuffer)

# in the rendering loop
gltools.drawVAO(vao, GL.GL_TRIANGLES)
```

The color of the sphere can be changed by calling `glColor*`:

```
glColor4f(1.0, 0.0, 0.0, 1.0) # red
gltools.drawVAO(vao, GL.GL_TRIANGLES)
```

Raw coordinates can be transformed prior to uploading to VBOs. Here we can rotate vertex positions and normals so the equator rests on Z-axis:

```
r = mt.rotationMatrix(90.0, (1.0, 0, 0.0)) # 90 degrees about +X axis
vertices = mt.applyMatrix(r, vertices)
normals = mt.applyMatrix(r, normals)
```

psychopy.tools.gltools.createPlane

`psychopy.tools.gltools.createPlane` (*size=1.0, 1.0*)

Create a plane.

Procedurally generate a plane (or quad) mesh by specifying its size. Texture coordinates are computed automatically, with origin at the bottom left of the plane. The generated plane is perpendicular to the +Z axis, origin of the plane is at its center.

Parameters **size** (*tuple or float*) – Dimensions of the plane. If a single value is specified, the plane will be square. Provide a tuple of floats to specify the width and length of the plane (eg. *size=(0.2, 1.3)*).

Returns Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

Return type `tuple`

Examples

Create a plane mesh and draw it:

```
vertices, textureCoords, normals, faces = gltools.createPlane()

vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(
    faces.flatten(),
    target=GL.GL_ELEMENT_ARRAY_BUFFER,
    dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO({0: vertexVBO, 8: texCoordVBO, 2: normalsVBO},
    indexBuffer=indexBuffer)

# in the rendering loop
gltools.drawVAO(vao, GL.GL_TRIANGLES)
```

psychoPy.tools.gltools.createMeshGridFromArrays

`psychoPy.tools.gltools.createMeshGridFromArrays` (*xvals*, *yvals*, *zvals=None*, *tessMode='diag'*, *computeNormals=True*)

Create a mesh grid using coordinates from arrays.

Generates a mesh using data in provided in 2D arrays of vertex coordinates. Triangle faces are automatically computed by this function by joining adjacent vertices at neighbouring indices in the array. Texture coordinates are generated covering the whole mesh, with origin at the bottom left.

Parameters

- **xvals** (*array_like*) – NxM arrays of X and Y coordinates. Both arrays must have the same shape. the resulting mesh will have a single vertex for each X and Y pair. Faces will be generated to connect adjacent coordinates in the array.
- **yvals** (*array_like*) – NxM arrays of X and Y coordinates. Both arrays must have the same shape. the resulting mesh will have a single vertex for each X and Y pair. Faces will be generated to connect adjacent coordinates in the array.
- **zvals** (*array_like, optional*) – NxM array of Z coordinates for each X and Y. Must have the same shape as X and Y. If not specified, the Z coordinates will be filled with zeros.
- **tessMode** (*str, optional*) – Tessellation mode. Specifies how faces are generated. Options are 'center', 'radial', and 'diag'. Default is 'diag'. Modes 'radial' and 'center' work best with odd numbered array dimensions.
- **computeNormals** (*bool, optional*) – Compute normals for the generated mesh. If *False*, all normals are set to face in the +Z direction. Presently, computing normals is a slow operation and may not be needed for some meshes.

Returns Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

Return type tuple

Examples

Create a 3D sine grating mesh using 2D arrays:

```
x = np.linspace(0, 1.0, 32)
y = np.linspace(1.0, 0.0, 32)
xx, yy = np.meshgrid(x, y)
zz = np.tile(np.sin(np.linspace(0.0, 32., 32)) * 0.02, (32, 1))

vertices, textureCoords, normals, faces = gltools.
↳createMeshGridFromArray(xx, yy, zz)
```

psychopy.tools.gltools.createMeshGrid

psychopy.tools.gltools.**createMeshGrid** (*size=1.0, 1.0, subdiv=0, tessMode='diag'*)

Create a grid mesh.

Procedurally generate a grid mesh by specifying its size and number of sub-divisions. Texture coordinates are computed automatically. The origin is at the center of the mesh. The generated grid is perpendicular to the +Z axis, origin of the grid is at its center.

Parameters

- **size** (*tuple or float*) – Dimensions of the mesh. If a single value is specified, the plane will be square. Provide a tuple of floats to specify the width and length of the plane (eg. *size=(0.2, 1.3)*).
- **subdiv** (*int, optional*) – Number of subdivisions. Zero subdivisions are applied by default, and the resulting mesh will only have vertices at the corners.
- **tessMode** (*str, optional*) – Tessellation mode. Specifies how faces are subdivided. Options are 'center', 'radial', and 'diag'. Default is 'diag'. Modes 'radial' and 'center' work best with an odd number of subdivisions.

Returns Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

Return type tuple

Examples

Create a grid mesh and draw it:

```
vertices, textureCoords, normals, faces = gltools.createPlane()

vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(
    faces.flatten(),
    target=GL.GL_ELEMENT_ARRAY_BUFFER,
    dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO({0: vertexVBO, 8: texCoordVBO, 2: normalsVBO},
    indexBuffer=indexBuffer)
```

(continues on next page)

(continued from previous page)

```
# in the rendering loop
gltools.drawVAO(vao, GL.GL_TRIANGLES)
```

Randomly displace vertices off the plane of the grid by setting the Z value per vertex:

```
vertices, textureCoords, normals, faces = gltools.
↳createMeshGrid(subdiv=11)

numVerts = vertices.shape[0]
vertices[:, 2] = np.random.uniform(-0.02, 0.02, (numVerts,)) # Z

# you must recompute surface normals to get correct shading!
normals = gltools.calculateVertexNormals(vertices, faces)

# create a VAO as shown in the previous example here to draw it ...
```

psychopy.tools.gltools.createBox

`psychopy.tools.gltools.createBox` (*size=1.0, 1.0, 1.0, flipFaces=False*)

Create a box mesh.

Create a box mesh by specifying its *size* in three dimensions (x, y, z), or a single value (*float*) to create a cube. The resulting box will be centered about the origin. Texture coordinates and normals are automatically generated for each face.

Setting *flipFaces=True* will make faces and normals point inwards, this allows boxes to be viewed and lit correctly from the inside.

Parameters

- **size** (*tuple or float*) – Dimensions of the mesh. If a single value is specified, the box will be a cube. Provide a tuple of floats to specify the width, length, and height of the box (eg. *size=(0.2, 1.3, 2.1)*).
- **flipFaces** (*bool, optional*) – If *True*, normals and face windings will be set to point inward towards the center of the box. Texture coordinates will remain the same. Default is *False*.

Returns Vertex attribute arrays (position, texture coordinates, and normals) and triangle indices.

Return type `tuple`

Examples

Create a box mesh and draw it:

```
vertices, textureCoords, normals, faces = gltools.createBox()

vertexVBO = gltools.createVBO(vertices)
texCoordVBO = gltools.createVBO(textureCoords)
normalsVBO = gltools.createVBO(normals)
indexBuffer = gltools.createVBO(
    faces.flatten(),
    target=GL.GL_ELEMENT_ARRAY_BUFFER,
```

(continues on next page)

(continued from previous page)

```

dataType=GL.GL_UNSIGNED_INT)

vao = gltools.createVAO({0: vertexVBO, 8: texCoordVBO, 2: normalsVBO},
    indexBuffer=indexBuffer)

# in the rendering loop
gltools.drawVAO(vao, GL.GL_TRIANGLES)
    
```

psychopy.tools.gltools.transformMeshPosOri

psychopy.tools.gltools.**transformMeshPosOri** (*vertices, normals, pos=0.0, 0.0, 0.0, ori=0.0, 0.0, 0.0, 1.0*)

Transform a mesh.

Transform mesh vertices and normals to a new position and orientation using a position coordinate and rotation quaternion. Values *vertices* and *normals* must be the same shape. This is intended to be used when editing raw vertex data prior to rendering. Do not use this to change the configuration of an object while rendering.

Parameters

- **vertices** (*array_like*) – Nx3 array of vertices.
- **normals** (*array_like*) – Nx3 array of normals.
- **pos** (*array_like, optional*) – Position vector to transform mesh vertices. If Nx3, *vertices* will be transformed by corresponding rows of *pos*.
- **ori** (*array_like, optional*) – Orientation quaternion in form [x, y, z, w]. If Nx4, *vertices* and *normals* will be transformed by corresponding rows of *ori*.

Returns Transformed vertices and normals.

Return type tuple

Examples

Create and re-orient a plane to face upwards:

```

vertices, textureCoords, normals, faces = createPlane()

# rotation quaternion
qr = quatFromAxisAngle((1., 0., 0.), -90.0) # -90 degrees about +X axis

# transform the normals and points
vertices, normals = transformMeshPosOri(vertices, normals, ori=qr)
    
```

Any *create** primitive generating function can be used in place of *createPlane*.

psychopy.tools.gltools.calculateVertexNormals

`psychopy.tools.gltools.calculateVertexNormals` (*vertices*, *faces*, *shading*='smooth')

Calculate vertex normals given vertices and triangle faces.

Finds all faces sharing a vertex index and sets its normal to either the face normal if *shading*='flat' or the average normals of adjacent faces if *shading*='smooth'. Flat shading only works correctly if each vertex belongs to exactly one face.

The direction of the normals are determined by the winding order of triangles, assumed counter clock-wise (OpenGL default). Most model editing software exports using this convention. If not, winding orders can be reversed by calling:

```
faces = np.fliplr(faces)
```

In some case, creases may appear if vertices are at the same location, but do not share the same index.

Parameters

- **vertices** (*array_like*) – Nx3 vertex positions.
- **faces** (*array_like*) – Nx3 vertex indices.
- **shading** (*str*, *optional*) – Shading mode. Options are 'smooth' and 'flat'. Flat only works with meshes where no vertex index is shared across faces.

Returns Vertex normals array with the same shape as *vertices*. Computed normals are normalized.

Return type ndarray

Examples

Recomputing vertex normals for a UV sphere:

```
# create a sphere and discard normals
vertices, textureCoords, _, faces = gltools.createUVSphere()
normals = gltools.calculateVertexNormals(vertices, faces)
```

Miscellaneous

Miscellaneous tools for working with OpenGL.

<code>getIntegerv(parName)</code>	Get a single integer parameter value, return it as a Python integer.
<code>getFloatv(parName)</code>	Get a single float parameter value, return it as a Python float.
<code>getString(parName)</code>	Get a single string parameter value, return it as a Python UTF-8 string.
<code>getOpenGLInfo()</code>	Get general information about the OpenGL implementation on this machine.
<code>getModelViewMatrix()</code>	Get the present model matrix from the OpenGL matrix stack.
<code>getProjectionMatrix()</code>	Get the present projection matrix from the OpenGL matrix stack.

psychopy.tools.gltools.getIntegerv

`psychopy.tools.gltools.getIntegerv(parName)`

Get a single integer parameter value, return it as a Python integer.

Parameters `pName` (*int*) – OpenGL property enum to query (e.g. `GL_MAJOR_VERSION`).

Returns

Return type `int`

psychopy.tools.gltools.getFloatv

`psychopy.tools.gltools.getFloatv(parName)`

Get a single float parameter value, return it as a Python float.

Parameters `pName` (*float*) – OpenGL property enum to query.

Returns

Return type `float`

psychopy.tools.gltools.getString

`psychopy.tools.gltools.getString(parName)`

Get a single string parameter value, return it as a Python UTF-8 string.

Parameters `pName` (*int*) – OpenGL property enum to query (e.g. `GL_VENDOR`).

Returns

Return type `str`

psychopy.tools.gltools.getOpenGLInfo

`psychopy.tools.gltools.getOpenGLInfo()`

Get general information about the OpenGL implementation on this machine. This should provide a consistent means of doing so regardless of the OpenGL interface we are using.

Returns are dictionary with the following fields:

```
vendor, renderer, version, majorVersion, minorVersion, doubleBuffer,
maxTextureSize, stereo, maxSamples, extensions
```

Supported extensions are returned as a list in the 'extensions' field. You can check if a platform supports an extension by checking the membership of the extension name in that list.

Returns

Return type `OpenGLInfo`

psychopy.tools.gltools.getModelViewMatrix

`psychopy.tools.gltools.getModelViewMatrix()`
 Get the present model matrix from the OpenGL matrix stack.

Returns 4x4 model/view matrix.

Return type ndarray

psychopy.tools.gltools.getProjectionMatrix

`psychopy.tools.gltools.getProjectionMatrix()`
 Get the present projection matrix from the OpenGL matrix stack.

Returns 4x4 projection matrix.

Return type ndarray

Examples

Working with Framebuffer Objects (FBOs):

Creating an empty framebuffer with no attachments:

```
fbo = createFBO() # invalid until attachments are added
```

Create a render target with multiple color texture attachments:

```
colorTex = createTexImage2D(1024,1024) # empty texture
depthRb = createRenderbuffer(800,600,internalFormat=GL.GL_DEPTH24_STENCIL8)

GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fbo.id)
attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
attach(GL.GL_STENCIL_ATTACHMENT, depthRb)
# or attach(GL.GL_DEPTH_STENCIL_ATTACHMENT, depthRb)
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, 0)
```

Attach FBO images using a context. This automatically returns to the previous FBO binding state when complete. This is useful if you don't know the current binding state:

```
with useFBO(fbo):
    attach(GL.GL_COLOR_ATTACHMENT0, colorTex)
    attach(GL.GL_DEPTH_ATTACHMENT, depthRb)
    attach(GL.GL_STENCIL_ATTACHMENT, depthRb)
```

How to set `userData` some custom function might access:

```
fbo.userData['flags'] = ['left_eye', 'clear_before_use']
```

Binding an FBO for drawing/reading:

```
GL.glBindFramebuffer(GL.GL_FRAMEBUFFER, fb.id)
```

Depth-only framebuffers are valid, sometimes need for generating shadows:

```
depthTex = createTexImage2D(800, 600,
                            internalFormat=GL.GL_DEPTH_COMPONENT24,
                            pixelFormat=GL.GL_DEPTH_COMPONENT)
fbo = createFBO([(GL.GL_DEPTH_ATTACHMENT, depthTex)])
```

Deleting a framebuffer when done with it. This invalidates the framebuffer's ID and makes it available for use:

```
deleteFBO(fbo)
```

9.7.5 `psychoPy.tools.imagetools`

Functions and classes related to image handling

<code>array2image(a)</code>	Takes an array and returns an image object (PIL).
<code>image2array(im)</code>	Takes an image object (PIL) and returns a numpy array.
<code>makeImageAuto(inarray)</code>	Combines float_uint8 and image2array operations ie.

Function details

`psychoPy.tools.imagetools.array2image(a)`

Takes an array and returns an image object (PIL).

`psychoPy.tools.imagetools.image2array(im)`

Takes an image object (PIL) and returns a numpy array.

`psychoPy.tools.imagetools.makeImageAuto(inarray)`

Combines float_uint8 and image2array operations ie. scales a numeric array from -1:1 to 0:255 and converts to PIL image format.

9.7.6 `psychoPy.tools.mathtools`

Assorted math functions for working with vectors, matrices, and quaternions. These functions are intended to provide basic support for common mathematical operations associated with displaying stimuli (e.g. animation, posing, rendering, etc.)

For tools related to view transformations, see `viewtools`.

Vectors

Tools for working with 2D and 3D vectors.

<code>length(v[, squared, out, dtype])</code>	Get the length of a vector.
<code>normalize(v[, out, dtype])</code>	Normalize a vector or quaternion.
<code>orthogonalize(v, n[, out, dtype])</code>	Orthogonalize a vector relative to a normal vector.
<code>reflect(v, n[, out, dtype])</code>	Reflection of a vector.
<code>dot(v0, v1[, out, dtype])</code>	Dot product of two vectors.
<code>cross(v0, v1[, out, dtype])</code>	Cross product of 3D vectors.
<code>project(v0, v1[, out, dtype])</code>	Project a vector onto another.
<code>perp(v, n[, norm, out, dtype])</code>	Project v to be a perpendicular axis of n .

continues on next page

Table 9.69 – continued from previous page

<code>lerp(v0, v1, t[, out, dtype])</code>	Linear interpolation (LERP) between two vectors/coordinates.
<code>distance(v0, v1[, out, dtype])</code>	Get the distance between vectors/coordinates.
<code>angleTo(v, point[, degrees, out, dtype])</code>	Get the relative angle to a point from a vector.
<code>bisector(v0, v1[, norm, out, dtype])</code>	Get the angle bisector.
<code>surfaceNormal(tri[, norm, out, dtype])</code>	Compute the surface normal of a given triangle.
<code>surfaceBitangent(tri, uv[, norm, out, dtype])</code>	Compute the bitangent vector of a given triangle.
<code>surfaceTangent(tri, uv[, norm, out, dtype])</code>	Compute the tangent vector of a given triangle.
<code>vertexNormal(faceNormals[, norm, out, dtype])</code>	Compute a vertex normal from shared triangles.
<code>fixTangentHandedness(tangents, normals, ...)</code>	Ensure the handedness of tangent vectors are all the same.
<code>ortho3Dto2D(p, orig, normal, up[, right, dtype])</code>	Get the planar coordinates of an orthogonal projection of a 3D point onto a 2D plane.
<code>transform(pos, ori, points[, out, dtype])</code>	Transform points using a position and orientation.
<code>scale(sf, points[, out, dtype])</code>	Scale points by a factor.

psychoPy.tools.mathtools.length

`psychoPy.tools.mathtools.length(v, squared=False, out=None, dtype=None)`

Get the length of a vector.

Parameters

- **v** (*array_like*) – Vector to normalize, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
- **squared** (*bool, optional*) – If True the squared length is returned. The default is False.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Length of vector *v*.

Return type float or ndarray

psychoPy.tools.mathtools.normalize

`psychoPy.tools.mathtools.normalize(v, out=None, dtype=None)`

Normalize a vector or quaternion.

v [array_like] Vector to normalize, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors. All vectors should have nonzero length.

out [ndarray, optional] Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.

dtype [dtype or str, optional] Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Normalized vector *v*.

Return type ndarray

Notes

- If the vector has length is zero, a vector of all zeros is returned after normalization.

Examples

Normalize a vector:

```
v = [1., 2., 3., 4.]
vn = normalize(v)
```

The *normalize* function is vectorized. It's considerably faster to normalize large arrays of vectors than to call *normalize* separately for each one:

```
v = np.random.uniform(-1.0, 1.0, (1000, 4,)) # 1000 length 4 vectors
vn = np.zeros((1000, 4)) # place to write values
normalize(v, out=vn) # very fast!

# don't do this!
for i in range(1000):
    vn[i, :] = normalize(v[i, :])
```

psychoPy.tools.mathtools.orthogonalize

psychoPy.tools.mathtools.orthogonalize(*v*, *n*, *out=None*, *dtype=None*)

Orthogonalize a vector relative to a normal vector.

This function ensures that *v* is perpendicular (or orthogonal) to *n*.

Parameters

- **v** (*array_like*) – Vector to orthogonalize, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
- **n** (*array_like*) – Normal vector, must have same shape as *v*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns Orthogonalized vector *v* relative to normal vector *n*.

Return type ndarray

Warning: If *v* and *n* are the same, the direction of the perpendicular vector is indeterminate. The resulting vector is degenerate (all zeros).

psychopy.tools.mathtools.reflect

`psychopy.tools.mathtools.reflect` (*v*, *n*, *out=None*, *dtype=None*)

Reflection of a vector.

Get the reflection of *v* relative to normal *n*.

Parameters

- **v** (*array_like*) – Vector to reflect, can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
- **n** (*array_like*) – Normal vector, must have same shape as *v*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Reflected vector *v* off normal *n*.

Return type ndarray

psychopy.tools.mathtools.dot

`psychopy.tools.mathtools.dot` (*v0*, *v1*, *out=None*, *dtype=None*)

Dot product of two vectors.

The behaviour of this function depends on the format of the input arguments:

- If *v0* and *v1* are 1D, the dot product is returned as a scalar and *out* is ignored.
- If *v0* and *v1* are 2D, a 1D array of dot products between corresponding row vectors are returned.
- If either *v0* and *v1* are 1D and 2D, an array of dot products between each row of the 2D vector and the 1D vector are returned.

Parameters

- **v0** (*array_like*) – Vector(s) to compute dot products of (e.g. [x, y, z]). *v0* must have equal or fewer dimensions than *v1*.
- **v1** (*array_like*) – Vector(s) to compute dot products of (e.g. [x, y, z]). *v0* must have equal or fewer dimensions than *v1*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Dot product(s) of *v0* and *v1*.

Return type ndarray

psychopy.tools.mathtools.cross

psychopy.tools.mathtools.**cross** (*v0*, *v1*, *out=None*, *dtype=None*)

Cross product of 3D vectors.

The behavior of this function depends on the dimensions of the inputs:

- If *v0* and *v1* are 1D, the cross product is returned as 1D vector.
- If *v0* and *v1* are 2D, a 2D array of cross products between corresponding row vectors are returned.
- If either *v0* and *v1* are 1D and 2D, an array of cross products between each row of the 2D vector and the 1D vector are returned.

Parameters

- **v0** (*array_like*) – Vector(s) in form [x, y, z] or [x, y, z, 1].
- **v1** (*array_like*) – Vector(s) in form [x, y, z] or [x, y, z, 1].
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Cross product of *v0* and *v1*.

Return type ndarray

Notes

- If input vectors are 4D, the last value of cross product vectors is always set to one.
- If input vectors *v0* and *v1* are Nx3 and *out* is Nx4, the cross product is computed and the last column of *out* is filled with ones.

Examples

Find the cross product of two vectors:

```
a = normalize([1, 2, 3])
b = normalize([3, 2, 1])
c = cross(a, b)
```

If input arguments are 2D, the function returns the cross products of corresponding rows:

```
# create two 6x3 arrays with random numbers
shape = (6, 3)
a = normalize(np.random.uniform(-1.0, 1.0, shape))
b = normalize(np.random.uniform(-1.0, 1.0, shape))
cprod = np.zeros(shape) # output has the same shape as inputs
cross(a, b, out=cprod)
```

If a 1D and 2D vector are specified, the cross product of each row of the 2D array and the 1D array is returned as a 2D array:

```
a = normalize([1, 2, 3])
b = normalize(np.random.uniform(-1.0, 1.0, (6, 3)))
cprod = np.zeros(a.shape)
cross(a, b, out=cprod)
```

psychoy.tools.mathtools.project

psychoy.tools.mathtools.**project** (*v0*, *v1*, *out=None*, *dtype=None*)

Project a vector onto another.

Parameters

- **v0** (*array_like*) – Vector can be Nx2, Nx3, or Nx4. If a 2D array is specified, rows are treated as separate vectors.
- **v1** (*array_like*) – Vector to project onto *v0*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Projection of vector *v0* on *v1*.

Return type ndarray or float

psychoy.tools.mathtools.perp

psychoy.tools.mathtools.**perp** (*v*, *n*, *norm=True*, *out=None*, *dtype=None*)

Project *v* to be a perpendicular axis of *n*.

Parameters

- **v** (*array_like*) – Vector to project [x, y, z], may be Nx3.
- **n** (*array_like*) – Normal vector [x, y, z], may be Nx3.
- **norm** (*bool*) – Normalize the resulting axis. Default is *True*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Perpendicular axis of *n* from *v*.

Return type ndarray

Examples

Determine the local *up* (y-axis) of a surface or plane given *normal*:

```
normal = [0., 0.70710678, 0.70710678]
up = [1., 0., 0.]

yaxis = perp(up, normal)
```

Do a cross product to get the x-axis perpendicular to both:

```
xaxis = cross(yaxis, normal)
```

psychoPy.tools.mathtools.lerp

psychoPy.tools.mathtools.**lerp** (*v0*, *v1*, *t*, *out=None*, *dtype=None*)

Linear interpolation (LERP) between two vectors/coordinates.

Parameters

- **v0** (*array_like*) – Initial vector/coordinate. Can be 2D where each row is a point.
- **v1** (*array_like*) – Final vector/coordinate. Must be the same shape as *v0*.
- **t** (*float*) – Interpolation weight factor [0, 1].
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Vector at *t* with same shape as *v0* and *v1*.

Return type ndarray

Examples

Find the coordinate of the midpoint between two vectors:

```
u = [0., 0., 0.]
v = [0., 0., 1.]
midpoint = lerp(u, v, 0.5) # 0.5 to interpolate half-way between points
```

psychoPy.tools.mathtools.distance

psychoPy.tools.mathtools.**distance** (*v0*, *v1*, *out=None*, *dtype=None*)

Get the distance between vectors/coordinates.

The behaviour of this function depends on the format of the input arguments:

- If *v0* and *v1* are 1D, the distance is returned as a scalar and *out* is ignored.
- If *v0* and *v1* are 2D, an array of distances between corresponding row vectors are returned.
- If either *v0* and *v1* are 1D and 2D, an array of distances between each row of the 2D vector and the 1D vector are returned.

Parameters

- **v0** (*array_like*) – Vectors to compute the distance between.
- **v1** (*array_like*) – Vectors to compute the distance between.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Distance between vectors *v0* and *v1*.

Return type ndarray

psychopy.tools.mathtools.angleTo

`psychopy.tools.mathtools.angleTo(v, point, degrees=True, out=None, dtype=None)`

Get the relative angle to a point from a vector.

The behaviour of this function depends on the format of the input arguments:

- If *v0* and *v1* are 1D, the angle is returned as a scalar and *out* is ignored.
- If *v0* and *v1* are 2D, an array of angles between corresponding row vectors are returned.
- If either *v0* and *v1* are 1D and 2D, an array of angles between each row of the 2D vector and the 1D vector are returned.

Parameters

- **v** (*array_like*) – Direction vector [x, y, z].
- **point** (*array_like*) – Point(s) to compute angle to from vector *v*.
- **degrees** (*bool, optional*) – Return the resulting angles in degrees. If *False*, angles will be returned in radians. Default is *True*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Distance between vectors *v0* and *v1*.

Return type ndarray

psychopy.tools.mathtools.bisector

psychopy.tools.mathtools.**bisector** (*v0*, *v1*, *norm=False*, *out=None*, *dtype=None*)

Get the angle bisector.

Computes a vector which bisects the angle between *v0* and *v1*. Input vectors *v0* and *v1* must be non-zero.

Parameters

- **v0** (*array_like*) – Vectors to bisect [x, y, z]. Must be non-zero in length and have the same shape. Inputs can be Nx3 where the bisector for corresponding rows will be returned.
- **v1** (*array_like*) – Vectors to bisect [x, y, z]. Must be non-zero in length and have the same shape. Inputs can be Nx3 where the bisector for corresponding rows will be returned.
- **norm** (*bool, optional*) – Normalize the resulting bisector. Default is *False*.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Bisecting vector [x, y, z].

Return type ndarray

psychopy.tools.mathtools.surfaceNormal

psychopy.tools.mathtools.**surfaceNormal** (*tri*, *norm=True*, *out=None*, *dtype=None*)

Compute the surface normal of a given triangle.

Parameters

- **tri** (*array_like*) – Triangle vertices as 2D (3x3) array [p0, p1, p2] where each vertex is a length 3 array [vx, xy, vz]. The input array can be 3D (Nx3x3) to specify multiple triangles.
- **norm** (*bool, optional*) – Normalize computed surface normals if *True*, default is *True*.
- **out** (*ndarray, optional*) – Optional output array. Must have one fewer dimensions than *tri*. The shape of the last dimension must be 3.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Surface normal of triangle *tri*.

Return type ndarray

Examples

Compute the surface normal of a triangle:

```
vertices = [[-1., 0., 0.], [0., 1., 0.], [1, 0, 0]]
norm = surfaceNormal(vertices)
```

Find the normals for multiple triangles, and put results in a pre-allocated array:

```
vertices = [[[ -1., 0., 0.], [0., 1., 0.], [1, 0, 0]], # 2x3x3
            [[ 1., 0., 0.], [0., 1., 0.], [-1, 0, 0]]]
normals = np.zeros((2, 3)) # normals from two triangles triangles
surfaceNormal(vertices, out=normals)
```

psychoPy.tools.mathtools.surfaceBitangent

`psychoPy.tools.mathtools.surfaceBitangent` (*tri, uv, norm=True, out=None, dtype=None*)

Compute the bitangent vector of a given triangle.

This function can be used to generate bitangent vertex attributes for normal mapping. After computing bitangents, one may orthogonalize them with vertex normals using the `orthogonalize()` function, or within the fragment shader. Uses texture coordinates at each triangle vertex to determine the direction of the vector.

Parameters

- **tri** (*array_like*) – Triangle vertices as 2D (3x3) array [p0, p1, p2] where each vertex is a length 3 array [vx, xy, vz]. The input array can be 3D (Nx3x3) to specify multiple triangles.
- **uv** (*array_like*) – Texture coordinates associated with each face vertex as a 2D array (3x2) where each texture coordinate is length 2 array [u, v]. The input array can be 3D (Nx3x2) to specify multiple texture coordinates if multiple triangles are specified.
- **norm** (*bool, optional*) – Normalize computed bitangents if True, default is True.
- **out** (*ndarray, optional*) – Optional output array. Must have one fewer dimensions than *tri*. The shape of the last dimension must be 3.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Surface bitangent of triangle *tri*.

Return type ndarray

Examples

Computing the bitangents for two triangles from vertex and texture coordinates (UVs):

```
# array of triangle vertices (2x3x3)
tri = np.asarray([
    [(-1.0, 1.0, 0.0), (-1.0, -1.0, 0.0), (1.0, -1.0, 0.0)], # 1
    [(-1.0, 1.0, 0.0), (-1.0, -1.0, 0.0), (1.0, -1.0, 0.0)]] # 2

# array of triangle texture coordinates (2x3x2)
uv = np.asarray([
```

(continues on next page)

(continued from previous page)

```

    [(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)], # 1
    [(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)]) # 2

bitangents = surfaceBitangent(tri, uv, norm=True) # bitangents (2x3)

```

psychopy.tools.mathtools.surfaceTangent

`psychopy.tools.mathtools.surfaceTangent` (*tri*, *uv*, *norm=True*, *out=None*, *dtype=None*)

Compute the tangent vector of a given triangle.

This function can be used to generate tangent vertex attributes for normal mapping. After computing tangents, one may orthogonalize them with vertex normals using the `orthogonalize()` function, or within the fragment shader. Uses texture coordinates at each triangle vertex to determine the direction of the vector.

Parameters

- **tri** (*array_like*) – Triangle vertices as 2D (3x3) array [p0, p1, p2] where each vertex is a length 3 array [vx, xy, vz]. The input array can be 3D (Nx3x3) to specify multiple triangles.
- **uv** (*array_like*) – Texture coordinates associated with each face vertex as a 2D array (3x2) where each texture coordinate is length 2 array [u, v]. The input array can be 3D (Nx3x2) to specify multiple texture coordinates if multiple triangles are specified. If so *N* must be the same size as the first dimension of *tri*.
- **norm** (*bool, optional*) – Normalize computed tangents if True, default is True.
- **out** (*ndarray, optional*) – Optional output array. Must have one fewer dimensions than *tri*. The shape of the last dimension must be 3.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns Surface normal of triangle *tri*.

Return type ndarray

Examples

Compute surface normals, tangents, and bitangents for a list of triangles:

```

# triangle vertices (2x3x3)
vertices = [[[-1., 0., 0.], [0., 1., 0.], [1, 0, 0]],
            [[1., 0., 0.], [0., 1., 0.], [-1, 0, 0]]]

# array of triangle texture coordinates (2x3x2)
uv = np.asarray([
    [(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)], # 1
    [(0.0, 1.0), (0.0, 0.0), (1.0, 0.0)]) # 2

normals = surfaceNormal(vertices)
tangents = surfaceTangent(vertices, uv)
bitangents = cross(normals, tangents) # or use `surfaceBitangent`

```

Orthogonalize a surface tangent with a vertex normal vector to get the vertex tangent and bitangent vectors:


```
vertexTangent = orthogonalize(faceTangent, vertexNormal)
vertexBitangent = cross(vertexTangent, vertexNormal)
```

Ensure computed vectors have the same handedness, if not, flip the tangent vector (important for applications like normal mapping):

```
# tangent, bitangent, and normal are 2D
tangent[dot(cross(normal, tangent), bitangent) < 0.0, :] *= -1.0
```

psychoPy.tools.mathtools.vertexNormal

psychoPy.tools.mathtools.**vertexNormal** (*faceNormals*, *norm=True*, *out=None*, *dtype=None*)

Compute a vertex normal from shared triangles.

This function computes a vertex normal by averaging the surface normals of the triangles it belongs to. If model has no vertex normals, first use `surfaceNormal()` to compute them, then run `vertexNormal()` to compute vertex normal attributes.

While this function is mainly used to compute vertex normals, it can also be supplied triangle tangents and bitangents.

Parameters

- **faceNormals** (*array_like*) – An array (Nx3) of surface normals.
- **norm** (*bool, optional*) – Normalize computed normals if True, default is True.
- **out** (*ndarray, optional*) – Optional output array.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Vertex normal.

Return type ndarray

Examples

Compute a vertex normal from the face normals of the triangles it belongs to:

```
normals = [[1., 0., 0.], [0., 1., 0.]] # adjacent face normals
vertexNorm = vertexNormal(normals)
```

psychoPy.tools.mathtools.fixTangentHandedness

psychoPy.tools.mathtools.**fixTangentHandedness** (*tangents*, *normals*, *bitangents*, *out=None*, *dtype=None*)

Ensure the handedness of tangent vectors are all the same.

Often 3D computed tangents may not have the same handedness due to how texture coordinates are specified. This function takes input surface vectors and ensures that tangents have the same handedness. Use this function if you notice that normal mapping shading appears reversed with respect to the incident light direction. The output array of corrected tangents can be used in place of the original.

Parameters

- **tangents** (*array_like*) – Input Nx3 arrays of triangle tangents, normals and bitangents. All arrays must have the same size.
- **normals** (*array_like*) – Input Nx3 arrays of triangle tangents, normals and bitangents. All arrays must have the same size.
- **bitangents** (*array_like*) – Input Nx3 arrays of triangle tangents, normals and bitangents. All arrays must have the same size.
- **out** (*ndarray, optional*) – Optional output array for tangents. If not specified, a new array of tangents will be allocated.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Array of tangents with handedness corrected.

Return type ndarray

psychopy.tools.mathtools.ortho3Dto2D

`psychopy.tools.mathtools.ortho3Dto2D(p, orig, normal, up, right=None, dtype=None)`

Get the planar coordinates of an orthogonal projection of a 3D point onto a 2D plane.

This function gets the nearest point on the plane which a 3D point falls on the plane.

Parameters

- **p** (*array_like*) – Point to be projected on the plane.
- **orig** (*array_like*) – Origin of the plane to test [x, y, z].
- **normal** (*array_like*) – Normal vector of the plane [x, y, z], must be normalized.
- **up** (*array_like*) – Normalized up (+Y) direction of the plane’s coordinate system. Must be perpendicular to *normal*.
- **right** (*array_like, optional*) – Perpendicular right (+X) axis. If not provided, the axis will be computed via the cross product between *normal* and *up*.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Coordinates on the plane [X, Y] where the 3D point projects towards perpendicularly.

Return type ndarray

Examples

This function can be used with `intersectRayPlane()` to find the location on the plane the ray intersects:

```
# plane information
planeOrigin = [0, 0, 0]
planeNormal = [0, 0, 1] # must be normalized
planeUpAxis = perp([0, 1, 0], planeNormal) # must also be normalized

# ray
rayDir = [0, 0, -1]
```

(continues on next page)

(continued from previous page)

```

rayOrigin = [0, 0, 5]

# get the intersect in 3D world space
pnt = intersectRayPlane(rayOrigin, rayDir, planeOrigin, planeNormal)

# get the 2D coordinates on the plane the intersect occurred
planeX, planeY = ortho3Dto2D(pnt, planeOrigin, planeNormal, planeUpAxis)
    
```

psychoPy.tools.mathtools.transform

psychoPy.tools.mathtools.**transform** (*pos, ori, points, out=None, dtype=None*)

Transform points using a position and orientation. Points are rotated then translated.

Parameters

- **pos** (*array_like*) – Position vector in form [x, y, z] or [x, y, z, 1].
- **ori** (*array_like*) – Orientation quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **points** (*array_like*) – Point(s) [x, y, z] to transform.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Transformed points.

Return type ndarray

Examples

Transform points by a position coordinate and orientation quaternion:

```

# rigid body pose
ori = quatFromAxisAngle([0., 0., -1.], 90.0, degrees=True)
pos = [0., 1.5, -3.]
# points to transform
points = np.array([[0., 1., 0., 1.], [-1., 0., 0., 1.]]) # [x, y, z, 1]
outPoints = np.zeros_like(points) # output array
transform(pos, ori, points, out=outPoints) # do the transformation
    
```

You can get the same results as the previous example using a matrix by doing the following:

```

R = rotationMatrix(90., [0., 0., -1])
T = translationMatrix([0., 1.5, -3.])
M = concatenate([R, T])
applyMatrix(M, points, out=outPoints)
    
```

If you are defining transformations with quaternions and coordinates, you can skip the costly matrix creation process by using *transform*.

Notes

- In performance tests, *applyMatrix* is noticeably faster than *transform* for very large arrays, however this is only true if you are applying the same transformation to all points.
- If the input arrays for *points* or *pos* is Nx4, the last column is ignored.

psychopy.tools.mathtools.scale

`psychopy.tools.mathtools.scale` (*sf*, *points*, *out=None*, *dtype=None*)

Scale points by a factor.

This is useful for converting points between units, and to stretch or compress points along a given axis. Scaling can be uniform which the same factor is applied along all axes, or anisotropic along specific axes.

Parameters

- **sf** (*array_like* or *float*) – Scaling factor. If scalar, all points will be scaled uniformly by that factor. If a vector, scaling will be anisotropic along an axis.
- **points** (*array_like*) – Point(s) [x, y, z] to scale.
- **out** (*ndarray*, *optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype* or *str*, *optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Scaled points.

Return type ndarray

Examples

Apply uniform scaling to points, here we scale to convert points in centimeters to meters:

```
CM_TO_METERS = 1.0 / 100.0
pointsCM = [[1, 2, 3], [4, 5, 6], [-1, 1, 0]]
pointsM = scale(CM_TO_METERS, pointsCM)
```

Anisotropic scaling along the X and Y axis:

```
pointsM = scale((SCALE_FACTOR_X, SCALE_FACTOR_Y), pointsCM)
```

Scale only on the X axis:

```
pointsM = scale((SCALE_FACTOR_X,), pointsCM)
```

Apply scaling on the Z axis only:

```
pointsM = scale((1.0, 1.0, SCALE_FACTOR_Z), pointsCM)
```

Quaternions

Tools for working with *quaternions*. Quaternions are used primarily here to represent rotations in 3D space.

<code>articulate(boneVecs, boneOris[, dtype])</code>	Articulate an armature.
<code>slerp(q0, q1, t[, shortest, out, dtype])</code>	Spherical linear interpolation (SLERP) between two quaternions.
<code>quatToAxisAngle(q[, degrees, dtype])</code>	Convert a quaternion to <i>axis</i> and <i>angle</i> representation.
<code>quatFromAxisAngle(axis, angle[, degrees, dtype])</code>	Create a quaternion to represent a rotation about <i>axis</i> vector by <i>angle</i> .
<code>quatYawPitchRoll(q[, degrees, out, dtype])</code>	Get the yaw, pitch, and roll of a quaternion's orientation relative to the world -Z axis.
<code>alignTo(v, t[, out, dtype])</code>	Compute a quaternion which rotates one vector to align with another.
<code>quatMagnitude(q[, squared, out, dtype])</code>	Get the magnitude of a quaternion.
<code>multQuat(q0, q1[, out, dtype])</code>	Multiply quaternion <i>q0</i> and <i>q1</i> .
<code>accumQuat(qlist[, out, dtype])</code>	Accumulate quaternion rotations.
<code>invertQuat(q[, out, dtype])</code>	Get the multiplicative inverse of a quaternion.
<code>applyQuat(q, points[, out, dtype])</code>	Rotate points/coordinates using a quaternion.
<code>quatToMatrix(q[, out, dtype])</code>	Create a 4x4 rotation matrix from a quaternion.

psychopy.tools.mathtools.articulate

`psychopy.tools.mathtools.articulate` (*boneVecs, boneOris, dtype=None*)

Articulate an armature.

This function is used for forward kinematics and posing by specifying a list of 'bones'. A bone has a length and orientation, where sequential bones are linked end-to-end. Returns the transformed origins of the bones in scene coordinates and their orientations.

There are many applications for forward kinematics such as posing armatures and stimuli for display (eg. mocap data). Another application is for getting the location of the end effector of coordinate measuring hardware, where encoders measure the joint angles and the length of linking members are known. This can be used for computing pose from "Sword of Damocles"¹ like hardware or some other haptic input devices which the participant wears (eg. a glove that measures joint angles in the hand). The computed pose of the joints can be used to interact with virtual stimuli.

Parameters

- **boneVecs** (*array_like*) – Bone lengths [x, y, z] as an Nx3 array.
- **boneOris** (*array_like*) – Orientation of the bones as quaternions in form [x, y, z, w], relative to the previous bone.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns Array of bone origins and orientations. The first origin is root position which is always at [0, 0, 0]. Use `transform()` to reposition the armature, or create a transformation matrix and use `applyMatrix` to translate and rotate the whole armature into position.

Return type tuple

¹ Sutherland, I. E. (1968). "A head-mounted three dimensional display". Proceedings of AFIPS 68, pp. 757-764

References

Examples

Compute the orientations and origins of segments of an arm:

```
# bone lengths
boneLengths = [[0., 1., 0.], [0., 1., 0.], [0., 1., 0.]]

# create quaternions for joints
shoulder = mt.quatFromAxisAngle('-y', 45.0)
elbow = mt.quatFromAxisAngle('+z', 45.0)
wrist = mt.quatFromAxisAngle('+z', 45.0)

# articulate the parts of the arm
boxPos, boxOri = mt.articulate(pos, [shoulder, elbow, wrist])

# assign positions and orientations to 3D objects
shoulderModel.thePose.posOri = (boxPos[0, :], boxOri[0, :])
elbowModel.thePose.posOri = (boxPos[1, :], boxOri[1, :])
wristModel.thePose.posOri = (boxPos[2, :], boxOri[2, :])
```

psychoPy.tools.mathtools.slerp

psychoPy.tools.mathtools.**slerp** (*q0, q1, t, shortest=True, out=None, dtype=None*)

Spherical linear interpolation (SLERP) between two quaternions.

The behaviour of this function depends on the types of arguments:

- If *q0* and *q1* are both 1-D and *t* is scalar, the interpolation at *t* is returned.
- If *q0* and *q1* are both 2-D Nx4 arrays and *t* is scalar, an Nx4 array is returned with each row containing the interpolation at *t* for each quaternion pair at matching row indices in *q0* and *q1*.

Parameters

- **q0** (*array_like*) – Initial quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **q1** (*array_like*) – Final quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **t** (*float*) – Interpolation weight factor within interval 0.0 and 1.0.
- **shortest** (*bool, optional*) – Ensure interpolation occurs along the shortest arc along the 4-D hypersphere (default is *True*).
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns Quaternion [x, y, z, w] at *t*.

Return type ndarray

Examples

Interpolate between two orientations:

```
q0 = quatFromAxisAngle(90.0, degrees=True)
q1 = quatFromAxisAngle(-90.0, degrees=True)
# halfway between 90 and -90 is 0.0 or quaternion [0. 0. 0. 1.]
qr = slerp(q0, q1, 0.5)
```

Example of smooth rotation of an object with fixed angular velocity:

```
degPerSec = 10.0 # rotate a stimulus at 10 degrees per second

# initial orientation, axis rotates in the Z direction
qr = quatFromAxisAngle([0., 0., -1.], 0.0, degrees=True)
# amount to rotate every second
qv = quatFromAxisAngle([0., 0., -1.], degPerSec, degrees=True)

# ---- within main experiment loop ----
# `frameTime` is the time elapsed in seconds from last `slerp`.
qr = multQuat(qr, slerp((0., 0., 0., 1.), qv, degPerSec * frameTime))
_, angle = quatToAxisAngle(qr) # discard axis, only need angle

# myStim is a GratingStim or anything with an 'ori' argument which
# accepts angle in degrees
myStim.ori = angle
myStim.draw()
```

psychopy.tools.mathtools.quatToAxisAngle

`psychopy.tools.mathtools.quatToAxisAngle` (*q*, *degrees=True*, *dtype=None*)

Convert a quaternion to *axis* and *angle* representation.

This allows you to use quaternions to set the orientation of stimuli that have an *ori* property.

Parameters

- **q** (*tuple*, *list* or *ndarray of float*) – Quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **degrees** (*bool*, *optional*) – Indicate *angle* is to be returned in degrees, otherwise *angle* will be returned in radians.
- **dtype** (*dtype* or *str*, *optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Axis and angle of quaternion in form ([ax, ay, az], angle). If *degrees* is *True*, the angle returned is in degrees, radians if *False*.

Return type `tuple`

Examples

Using a quaternion to rotate a stimulus a fixed angle each frame:

```
# initial orientation, axis rotates in the Z direction
qr = quatFromAxisAngle([0., 0., -1.], 0.0, degrees=True)
# rotation per-frame, here it's 0.1 degrees per frame
qf = quatFromAxisAngle([0., 0., -1.], 0.1, degrees=True)

# ---- within main experiment loop ----
# myStim is a GratingStim or anything with an 'ori' argument which
# accepts angle in degrees
qr = multQuat(qr, qf) # cumulative rotation
_, angle = quatToAxisAngle(qr) # discard axis, only need angle
myStim.ori = angle
myStim.draw()
```

psychoPy.tools.mathtools.quatFromAxisAngle

`psychoPy.tools.mathtools.quatFromAxisAngle` (*axis*, *angle*, *degrees=True*, *dtype=None*)
 Create a quaternion to represent a rotation about *axis* vector by *angle*.

Parameters

- **axis** (*tuple*, *list*, *ndarray* or *str*) – Axis vector components or axis name. If a vector, input must be length 3 [x, y, z]. A string can be specified for rotations about world axes (eg. '+x', '-z', '+y', etc.)
- **angle** (*float*) – Rotation angle in radians (or degrees if *degrees* is *True*. Rotations are right-handed about the specified *axis*.)
- **degrees** (*bool*, *optional*) – Indicate *angle* is in degrees, otherwise *angle* will be treated as radians.
- **dtype** (*dtype* or *str*, *optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns Quaternion [x, y, z, w].

Return type ndarray

Examples

Create a quaternion from specified *axis* and *angle*:

```
axis = [0., 0., -1.] # rotate about -Z axis
angle = 90.0 # angle in degrees
ori = quatFromAxisAngle(axis, angle, degrees=True) # using degrees!
```


psychopy.tools.mathtools.quatYawPitchRoll

`psychopy.tools.mathtools.quatYawPitchRoll` (*q*, *degrees=True*, *out=None*, *dtype=None*)

Get the yaw, pitch, and roll of a quaternion's orientation relative to the world -Z axis.

You can multiply the quaternion by the inverse of some other one to make the returned values referenced to a local coordinate system.

Parameters

- **q** (*tuple*, *list* or *ndarray of float*) – Quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **degrees** (*bool*, *optional*) – Indicate angles are to be returned in degrees, otherwise they will be returned in radians.
- **out** (*ndarray*) – Optional output array. Must have same *shape* and *dtype* as what is expected to be returned by this function if *out* was not specified.
- **dtype** (*dtype* or *str*, *optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns Yaw, pitch and roll [yaw, pitch, roll] of quaternion *q*.

Return type ndarray

psychopy.tools.mathtools.alignTo

`psychopy.tools.mathtools.alignTo` (*v*, *t*, *out=None*, *dtype=None*)

Compute a quaternion which rotates one vector to align with another.

Parameters

- **v** (*array_like*) – Vector [x, y, z] to rotate. Can be Nx3, but must have the same shape as *t*.
- **t** (*array_like*) – Target [x, y, z] vector to align to. Can be Nx3, but must have the same shape as *v*.
- **out** (*ndarray*, *optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype* or *str*, *optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns Quaternion which rotates *v* to *t*.

Return type ndarray

Examples

Rotate some vectors to align with other vectors, inputs should be normalized:

```
vec = [[1, 0, 0], [0, 1, 0], [1, 0, 0]]
targets = [[0, 1, 0], [0, -1, 0], [-1, 0, 0]]

qr = alignTo(vec, targets)
vecRotated = applyQuat(qr, vec)

numpy.allclose(vecRotated, targets) # True
```

Get matrix which orients vertices towards a point:

```
point = [5, 6, 7]
vec = [0, 0, -1] # initial facing is -Z (forward in GL)

targetVec = normalize(point - vec)
qr = alignTo(vec, targetVec) # get rotation to align

M = quatToMatrix(qr) # 4x4 transformation matrix
```

psychoPy.tools.mathtools.quatMagnitude

`psychoPy.tools.mathtools.quatMagnitude` (*q*, *squared=False*, *out=None*, *dtype=None*)

Get the magnitude of a quaternion.

A quaternion is normalized if its magnitude is 1.

Parameters

- **q** (*array_like*) – Quaternion(s) in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **squared** (*bool, optional*) – If True return the squared magnitude. If you are just checking if a quaternion is normalized, the squared magnitude will suffice to avoid the square root operation.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Magnitude of quaternion *q*.

Return type `float` or `ndarray`

psychoPy.tools.mathtools.multQuat

psychoPy.tools.mathtools.**multQuat** (*q0, q1, out=None, dtype=None*)

Multiply quaternion *q0* and *q1*.

The orientation of the returned quaternion is the combination of the input quaternions.

Parameters

- **q0** (*array_like*) – Quaternions to multiply in form [x, y, z, w] where w is real and x, y, z are imaginary components. If 2D (Nx4) arrays are specified, quaternions are multiplied row-wise between each array.
- **q1** (*array_like*) – Quaternions to multiply in form [x, y, z, w] where w is real and x, y, z are imaginary components. If 2D (Nx4) arrays are specified, quaternions are multiplied row-wise between each array.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Combined orientations of *q0* and *q1*.

Return type ndarray

Notes

- Quaternions are normalized prior to multiplication.

Examples

Combine the orientations of two quaternions:

```
a = quatFromAxisAngle([0, 0, -1], 45.0, degrees=True)
b = quatFromAxisAngle([0, 0, -1], 90.0, degrees=True)
c = multQuat(a, b) # rotates 135 degrees about -Z axis
```

psychoPy.tools.mathtools.accumQuat

psychoPy.tools.mathtools.**accumQuat** (*qlist, out=None, dtype=None*)

Accumulate quaternion rotations.

Chain multiplies an Nx4 array of quaternions, accumulating their rotations. This function can be used for computing the orientation of joints in an armature for forward kinematics. The first quaternion is treated as the ‘root’ and the last is the orientation of the end effector.

Parameters

- **q** (*array_like*) – Nx4 array of quaternions to accumulate, where each row is a quaternion.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified. In this case, the same shape as *qlist*.

- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Nx4 array of quaternions.

Return type ndarray

Examples

Get the orientation of joints in an armature if we know their relative angles:

```
shoulder = quatFromAxisAngle('-x', 45.0) # rotate shoulder down 45 deg
elbow = quatFromAxisAngle('+x', 45.0) # rotate elbow up 45 deg
wrist = quatFromAxisAngle('-x', 45.0) # rotate wrist down 45 deg
finger = quatFromAxisAngle('+x', 0.0) # keep finger in-line with wrist

armRotations = accumQuat([shoulder, elbow, wrist, finger])
```

psychopy.tools.mathtools.invertQuat

psychopy.tools.mathtools.**invertQuat** (*q, out=None, dtype=None*)

Get the multiplicative inverse of a quaternion.

This gives a quaternion which rotates in the opposite direction with equal magnitude. Multiplying a quaternion by its inverse returns an identity quaternion as both orientations cancel out.

Parameters

- **q** (*ndarray, list, or tuple of float*) – Quaternion to invert in form [x, y, z, w] where w is real and x, y, z are imaginary components. If *q* is 2D (Nx4), each row is treated as a separate quaternion and inverted.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Inverse of quaternion *q*.

Return type ndarray

Examples

Show that multiplying a quaternion by its inverse returns an identity quaternion where [x=0, y=0, z=0, w=1]:

```
angle = 90.0
axis = [0., 0., -1.]
q = quatFromAxisAngle(axis, angle, degrees=True)
qinv = invertQuat(q)
qr = multQuat(q, qinv)
qi = np.array([0., 0., 0., 1.]) # identity quaternion
print(np.allclose(qi, qr)) # True
```

Notes

- Quaternions are normalized prior to inverting.

psychopy.tools.mathtools.applyQuat

`psychopy.tools.mathtools.applyQuat` (*q*, *points*, *out=None*, *dtype=None*)

Rotate points/coordinates using a quaternion.

This is similar to using `applyMatrix` with a rotation matrix. However, it is computationally less intensive to use `applyQuat` if one only wishes to rotate points.

Parameters

- **q** (*array_like*) – Quaternion to invert in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **points** (*array_like*) – 2D array of vectors or points to transform, where each row is a single point. Only the x, y, and z components (the first three columns) are rotated. Additional columns are copied.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Transformed points.

Return type ndarray

Examples

Rotate points using a quaternion:

```
points = [[1., 0., 0.], [0., -1., 0.]]
quat = quatFromAxisAngle(-90.0, [0., 0., -1.], degrees=True)
pointsRotated = applyQuat(quat, points)
# [[0. 1. 0.]
#   [1. 0. 0.]
```

Show that you get the same result as a rotation matrix:

```
axis = [0., 0., -1.]
angle = -90.0
rotMat = rotationMatrix(axis, angle)[:3, :3] # rotation sub-matrix only
rotQuat = quatFromAxisAngle(angle, axis, degrees=True)
points = [[1., 0., 0.], [0., -1., 0.]]
isClose = np.allclose(applyMatrix(rotMat, points), # True
                      applyQuat(rotQuat, points))
```

Specifying an array to *q* where each row is a quaternion transforms points in corresponding rows of *points*:

```
points = [[1., 0., 0.], [0., -1., 0.]]
quats = [quatFromAxisAngle(-90.0, [0., 0., -1.], degrees=True),
         quatFromAxisAngle(45.0, [0., 0., -1.], degrees=True)]
applyQuat(quats, points)
```

psychoPy.tools.mathtools.quatToMatrix

psychoPy.tools.mathtools.**quatToMatrix**(*q*, *out=None*, *dtype=None*)

Create a 4x4 rotation matrix from a quaternion.

Parameters

- **q** (*tuple*, *list* or *ndarray of float*) – Quaternion to convert in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **out** (*ndarray* or *None*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype* or *str*, *optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns 4x4 rotation matrix in row-major order.

Return type ndarray or None

Examples

Convert a quaternion to a rotation matrix:

```
point = [0., 1., 0., 1.] # 4-vector form [x, y, z, 1.0]
ori = [0., 0., 0., 1.]
rotMat = quatToMatrix(ori)
# rotate 'point' using matrix multiplication
newPoint = np.matmul(rotMat.T, point) # returns [-1., 0., 0., 1.]
```

Rotate all points in an array (each row is a coordinate):

```
points = np.asarray([[0., 0., 0., 1.],
                    [0., 1., 0., 1.],
                    [1., 1., 0., 1.]])
newPoints = points.dot(rotMat)
```

Notes

- Quaternions are normalized prior to conversion.

Matrices

Tools to creating and using affine transformation matrices.

<code>matrixToQuat(m[, out, dtype])</code>	Convert a rotation matrix to a quaternion.
<code>matrixFromEulerAngles(rx, ry, rz[, degrees, ...])</code>	Construct a 4x4 rotation matrix from Euler angles.
<code>scaleMatrix(s[, out, dtype])</code>	Create a scaling matrix.
<code>rotationMatrix(angle[, axis, out, dtype])</code>	Create a rotation matrix.
<code>translationMatrix(t[, out, dtype])</code>	Create a translation matrix.
<code>invertMatrix(m[, out, dtype])</code>	Invert a square matrix.
<code>isOrthogonal(m)</code>	Check if a square matrix is orthogonal.

continues on next page

Table 9.71 – continued from previous page

<code>isAffine(m)</code>	Check if a 4x4 square matrix describes an affine transformation.
<code>multMatrix(matrices[, reverse, out, dtype])</code>	Chain multiplication of two or more matrices.
<code>concatenate(matrices[, out, dtype])</code>	Concatenate matrix transformations.
<code>normalMatrix(modelMatrix[, out, dtype])</code>	Get the normal matrix from a model matrix.
<code>forwardProject(objPos, modelView, proj[, ...])</code>	Project a point in a scene to a window coordinate.
<code>reverseProject(winPos, modelView, proj[, ...])</code>	Unproject window coordinates into object or scene coordinates.
<code>applyMatrix(m, points[, out, dtype])</code>	Apply a matrix over a 2D array of points.
<code>posOriToMatrix(pos, ori[, out, dtype])</code>	Convert a rigid body pose to a 4x4 transformation matrix.

psychoPy.tools.mathtools.matrixToQuat

`psychoPy.tools.mathtools.matrixToQuat` (*m*, *out=None*, *dtype=None*)

Convert a rotation matrix to a quaternion.

Parameters

- **m** (*array_like*) – 3x3 rotation matrix (row-major). A 4x4 affine transformation matrix may be provided, assuming the top-left 3x3 sub-matrix is orthonormal and is a rotation group.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Rotation quaternion.

Return type ndarray

Notes

- Depending on the input, returned quaternions may not be exactly the same as the one used to construct the rotation matrix (i.e. by calling `quatToMatrix`), typically when a large rotation angle is used. However, the returned quaternion should result in the same rotation when applied to points.

Examples

Converting a rotation matrix from the OpenGL matrix stack to a quaternion:

```
glRotatef(45., -1, 0, 0)

m = np.zeros((4, 4), dtype='float32') # store the matrix
GL.glGetFloatv(
    GL.GL_MODELVIEW_MATRIX,
    m.ctypes.data_as(ctypes.POINTER(ctypes.c_float)))

qr = matrixToQuat(m.T) # must be transposed
```

Interpolation between two 4x4 transformation matrices:

```

interpWeight = 0.5

posStart = mStart[:3, 3]
oriStart = matrixToQuat(mStart)

posEnd = mEnd[:3, 3]
oriEnd = matrixToQuat(mEnd)

oriInterp = slerp(qStart, qEnd, interpWeight)
posInterp = lerp(posStart, posEnd, interpWeight)

mInterp = posOriToMatrix(posInterp, oriInterp)
    
```

psychoPy.tools.mathtools.matrixFromEulerAngles

psychoPy.tools.mathtools.**matrixFromEulerAngles**(*rx*, *ry*, *rz*, *degrees=True*, *out=None*, *dtype=None*)

Construct a 4x4 rotation matrix from Euler angles.

Rotations are combined by first rotating about the X axis, then Y, and finally Z.

Parameters

- **rx** (*float*) – Rotation angles (pitch, yaw, and roll).
- **ry** (*float*) – Rotation angles (pitch, yaw, and roll).
- **rz** (*float*) – Rotation angles (pitch, yaw, and roll).
- **degrees** (*bool*, *optional*) – Rotation angles are specified in degrees. If *False*, they will be assumed as radians. Default is *True*.
- **out** (*ndarray*, *optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str*, *optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns 4x4 rotation matrix.

Return type ndarray

Examples

Demonstration of how a combination of axis-angle rotations is equivalent to a single call of *matrixFromEulerAngles*:

```

m1 = matrixFromEulerAngles(90., 45., 135.)

# construct rotation matrix from 3 orthogonal rotations
rx = rotationMatrix(90., (1, 0, 0)) # x-axis
ry = rotationMatrix(45., (0, 1, 0)) # y-axis
rz = rotationMatrix(135., (0, 0, 1)) # z-axis
m2 = concatenate([rz, ry, rx]) # note the order

print(numpy.allclose(m1, m2)) # True
    
```


Not only does `matrixFromEulerAngles` require less code, it also is considerably more efficient than constructing and multiplying multiple matrices.

psychopy.tools.mathtools.scaleMatrix

`psychopy.tools.mathtools.scaleMatrix` (*s*, *out=None*, *dtype=None*)

Create a scaling matrix.

The resulting matrix is the same as a generated by a `glScale` call.

Parameters

- **s** (*array_like*, *float* or *int*) – Scaling factor(s). If *s* is scalar (float), scaling will be uniform. Providing a vector of scaling values [*sx*, *sy*, *sz*] will result in an anisotropic scaling matrix if any of the values differ.
- **out** (*ndarray*, *optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype* or *str*, *optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns 4x4 scaling matrix in row-major order.

Return type ndarray

psychopy.tools.mathtools.rotationMatrix

`psychopy.tools.mathtools.rotationMatrix` (*angle*, *axis=0.0, 0.0, -1.0*, *out=None*, *dtype=None*)

Create a rotation matrix.

The resulting matrix will rotate points about *axis* by *angle*. The resulting matrix is similar to that produced by a `glRotate` call.

Parameters

- **angle** (*float*) – Rotation angle in degrees.
- **axis** (*array_like* or *str*) – Axis vector components or axis name. If a vector, input must be length 3. A string can be specified for rotations about world axes (eg. ‘+x’, ‘-z’, ‘+y’, etc.)
- **out** (*ndarray*, *optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype* or *str*, *optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns 4x4 scaling matrix in row-major order. Will be the same array as *out* if specified, if not, a new array will be allocated.

Return type ndarray

Notes

- Vector *axis* is normalized before creating the matrix.

psychoPy.tools.mathtools.translationMatrix

`psychoPy.tools.mathtools.translationMatrix` (*t*, *out=None*, *dtype=None*)

Create a translation matrix.

The resulting matrix is the same as generated by a `glTranslate` call.

Parameters

- **t** (*ndarray*, *tuple*, or *list of float*) – Translation vector [tx, ty, tz].
- **out** (*ndarray*, *optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype* or *str*, *optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns 4x4 translation matrix in row-major order. Will be the same array as *out* if specified, if not, a new array will be allocated.

Return type ndarray

psychoPy.tools.mathtools.invertMatrix

`psychoPy.tools.mathtools.invertMatrix` (*m*, *out=None*, *dtype=None*)

Invert a square matrix.

Parameters

- **m** (*array_like*) – Square matrix to invert. Inputs can be 4x4, 3x3 or 2x2.
- **out** (*ndarray*, *optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype* or *str*, *optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Matrix which is the inverse of *m*

Return type ndarray

psychoPy.tools.mathtools.isOrthogonal

`psychoPy.tools.mathtools.isOrthogonal` (*m*)

Check if a square matrix is orthogonal.

If a matrix is orthogonal, its columns form an orthonormal basis and is non-singular. An orthogonal matrix is invertible by simply taking the transpose of the matrix.

Parameters **m** (*array_like*) – Square matrix, either 2x2, 3x3 or 4x4.

Returns *True* if the matrix is orthogonal.

Return type `bool`

`psychopy.tools.mathtools.isAffine`

`psychopy.tools.mathtools.isAffine(m)`

Check if a 4x4 square matrix describes an affine transformation.

Parameters `m` (*array_like*) – 4x4 transformation matrix.

Returns `True` if the matrix is affine.

Return type `bool`

`psychopy.tools.mathtools.multMatrix`

`psychopy.tools.mathtools.multMatrix(matrices, reverse=False, out=None, dtype=None)`

Chain multiplication of two or more matrices.

Multiply a sequence of matrices together, reducing to a single product matrix. For instance, specifying *matrices* the sequence of matrices (A, B, C, D) will return the product (((AB)C)D). If *reverse=True*, the product will be (A(B(CD))).

Alternatively, a 3D array can be specified to *matrices* as a stack, where an index along axis 0 references a 2D slice storing matrix values. The product of the matrices along the axis will be returned. This is a bit more efficient than specifying separate matrices in a sequence, but the difference is negligible when only a few matrices are being multiplied.

Parameters

- **matrices** (*list, tuple or ndarray*) – Sequence or stack of matrices to multiply. All matrices must have the same dimensions.
- **reverse** (*bool, optional*) – Multiply matrices right-to-left. This is useful when dealing with transformation matrices, where the order of operations for transforms will appear the same as the order the matrices are specified. Default is ‘False’. When `True`, this function behaves similarly to `concatenate()`.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Matrix product.

Return type `ndarray`

Notes

- You may use `numpy.matmul` when dealing with only two matrices instead of `multMatrix`.
- If a single matrix is specified, the returned product will have the same values.

Examples

Chain multiplication of SRT matrices:

```
translate = translationMatrix((0.035, 0, -0.5))
rotate = rotationMatrix(90.0, (0, 1, 0))
scale = scaleMatrix(2.0)

SRT = multMatrix((translate, rotate, scale))
```

Same as above, but matrices are in a 3x4x4 array:

```
matStack = np.array((translate, rotate, scale))

# or ...
# matStack = np.zeros((3, 4, 4))
# matStack[0, :, :] = translate
# matStack[1, :, :] = rotate
# matStack[2, :, :] = scale

SRT = multMatrix(matStack)
```

Using `reverse=True` allows you to specify transformation matrices in the order which they will be applied:

```
SRT = multMatrix(np.array((scale, rotate, translate)), reverse=True)
```

psychopy.tools.mathtools.concatenate

`psychopy.tools.mathtools.concatenate` (*matrices*, *out=None*, *dtype=None*)

Concatenate matrix transformations.

Chain multiply matrices describing transform operations into a single matrix product, that when applied, transforms points and vectors with each operation in the order they're specified.

Parameters

- **matrices** (*list or tuple*) – List of matrices to concatenate. All matrices must all have the same size, usually 4x4 or 3x3.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns Matrix product.

Return type ndarray

See also:

- `multMatrix` : Chain multiplication of matrices.

Notes

- This function should only be used for combining transformation matrices. Use `multMatrix` for general matrix chain multiplication.

Examples

Create an SRT (scale, rotate, and translate) matrix to convert model-space coordinates to world-space:

```
S = scaleMatrix([2.0, 2.0, 2.0]) # scale model 2x
R = rotationMatrix(-90., [0., 0., -1]) # rotate -90 about -Z axis
T = translationMatrix([0., 0., -5.]) # translate point 5 units away

# product matrix when applied to points will scale, rotate and transform
# in that order.
SRT = concatenate([S, R, T])

# transform a point in model-space coordinates to world-space
pointModel = np.array([0., 1., 0., 1.])
pointWorld = np.matmul(SRT, pointModel.T) # point in WCS
# ... or ...
pointWorld = matrixApply(SRT, pointModel)
```

Create a model-view matrix from a world-space pose represented by an orientation (quaternion) and position (vector). The resulting matrix will transform model-space coordinates to eye-space:

```
# eye pose as quaternion and vector
stimOri = quatFromAxisAngle([0., 0., -1.], -45.0)
stimPos = [0., 1.5, -5.]

# create model matrix
R = quatToMatrix(stimOri)
T = translationMatrix(stimPos)
M = concatenate(R, T) # model matrix

# create a view matrix, can also be represented as 'pos' and 'ori'
eyePos = [0., 1.5, 0.]
eyeFwd = [0., 0., -1.]
eyeUp = [0., 1., 0.]
V = lookAt(eyePos, eyeFwd, eyeUp) # from viewtools

# modelview matrix
MV = concatenate([M, V])
```

You can put the created matrix in the OpenGL matrix stack as shown below. Note that the matrix must have a 32-bit floating-point data type and needs to be loaded transposed since OpenGL takes matrices in column-major order:

```
GL.glMatrixMode(GL.GL_MODELVIEW)

# pygamelet
MV = np.asarray(MV, dtype='float32') # must be 32-bit float!
ptrMV = MV.ctypes.data_as(ctypes.POINTER(ctypes.c_float))
```

(continues on next page)

(continued from previous page)

```
GL.glLoadTransposeMatrixf(ptrMV)

# PyOpenGL
MV = np.asarray(MV, dtype='float32')
GL.glLoadTransposeMatrixf(MV)
```

Furthermore, you can convert a point from model-space to homogeneous clip-space by concatenating the projection, view, and model matrices:

```
# compute projection matrix, functions here are from 'viewtools'
screenWidth = 0.52
screenAspect = w / h
scrDistance = 0.55
frustum = computeFrustum(screenWidth, screenAspect, scrDistance)
P = perspectiveProjectionMatrix(*frustum)

# multiply model-space points by MVP to convert them to clip-space
MVP = concatenate([M, V, P])
pointModel = np.array([0., 1., 0., 1.])
pointClipSpace = np.matmul(MVP, pointModel.T)
```

psychoPy.tools.mathtools.normalMatrix

`psychoPy.tools.mathtools.normalMatrix` (*modelMatrix*, *out=None*, *dtype=None*)

Get the normal matrix from a model matrix.

Parameters

- **modelMatrix** (*array_like*) – 4x4 homogeneous model matrix.
- **out** (*ndarray*, *optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str*, *optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Normal matrix.

Return type ndarray

psychoPy.tools.mathtools.forwardProject

`psychoPy.tools.mathtools.forwardProject` (*objPos*, *modelView*, *proj*, *viewport=None*, *out=None*, *dtype=None*)

Project a point in a scene to a window coordinate.

This function is similar to *gluProject* and can be used to find the window coordinate which a point projects to.

Parameters

- **objPos** (*array_like*) – Object coordinates (x, y, z). If an Nx3 array of coordinates is specified, where each row contains a window coordinate this function will return an array of projected coordinates with the same size.
- **modelView** (*array_like*) – 4x4 combined model and view matrix for returned value to be object coordinates. Specify only the view matrix for a coordinate in the scene.

- **proj** (*array_like*) – 4x4 projection matrix used for rendering.
- **viewport** (*array_like*) – Viewport rectangle for the window [x, y, w, h]. If not specified, the returned values will be in normalized device coordinates.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Normalized device or viewport coordinates [x, y, z] of the point. The z component is similar to the depth buffer value for the object point.

Return type ndarray

psychopy.tools.mathtools.reverseProject

`psychopy.tools.mathtools.reverseProject(winPos, modelView, proj, viewport=None, out=None, dtype=None)`

Unproject window coordinates into object or scene coordinates.

This function works like *gluUnProject* and can be used to find to an object or scene coordinate at the point on-screen (mouse coordinate or pixel). The coordinate can then be used to create a direction vector from the viewer’s eye location. Another use of this function is to convert depth buffer samples to object or scene coordinates. This is the inverse operation of *forwardProject()*.

Parameters

- **winPos** (*array_like*) – Window coordinates (x, y, z). If *viewport* is not specified, these should be normalized device coordinates. If an Nx3 array of coordinates is specified, where each row contains a window coordinate this function will return an array of unprojected coordinates with the same size. Usually, you only need to specify the x and y coordinate, leaving z as zero. However, you can specify z if sampling from a depth map or buffer to convert a depth sample to an actual location.
- **modelView** (*array_like*) – 4x4 combined model and view matrix for returned value to be object coordinates. Specify only the view matrix for a coordinate in the scene.
- **proj** (*array_like*) – 4x4 projection matrix used for rendering.
- **viewport** (*array_like*) – Viewport rectangle for the window [x, y, w, h]. Do not specify one if *winPos* is in already in normalized device coordinates.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Object or scene coordinates.

Return type ndarray

psychopy.tools.mathtools.applyMatrix

psychopy.tools.mathtools.**applyMatrix** (*m*, *points*, *out=None*, *dtype=None*)

Apply a matrix over a 2D array of points.

This function behaves similarly to the following *Numpy* statement:

```
points[:, :] = points.dot(m.T)
```

Transformation matrices specified to *m* must have dimensions 4x4, 3x4, 3x3 or 2x2. With the exception of 4x4 matrices, input *points* must have the same number of columns as the matrix has rows. 4x4 matrices can be used to transform both Nx4 and Nx3 arrays.

Parameters

- **m** (*array_like*) – Matrix with dimensions 2x2, 3x3, 3x4 or 4x4.
- **points** (*array_like*) – 2D array of points/coordinates to transform. Each row should have length appropriate for the matrix being used.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Transformed coordinates.

Return type ndarray

Notes

- Input (*points*) and output (*out*) arrays cannot be the same instance for this function.
- In the case of 4x4 input matrices, this function performs optimizations based on whether the input matrix is affine, greatly improving performance when working with Nx3 arrays.

Examples

Construct a matrix and transform a point:

```
# identity 3x3 matrix for this example
M = [[1.0, 0.0, 0.0],
      [0.0, 1.0, 0.0],
      [0.0, 0.0, 1.0]]

pnt = [1.0, 0.0, 0.0]

pntNew = applyMatrix(M, pnt)
```

Construct an SRT matrix (scale, rotate, transform) and transform an array of points:

```
S = scaleMatrix([5.0, 5.0, 5.0]) # scale 5x
R = rotationMatrix(180., [0., 0., -1]) # rotate 180 degrees
T = translationMatrix([0., 1.5, -3.]) # translate point up and away
M = concatenate([S, R, T]) # create transform matrix
```

(continues on next page)

(continued from previous page)

```
# points to transform
points = np.array([[0., 1., 0., 1.], [-1., 0., 0., 1.]]) # [x, y, z, w]
newPoints = applyMatrix(M, points) # apply the transformation
```

Convert CIE-XYZ colors to sRGB:

```
sRGBMatrix = [[3.2404542, -1.5371385, -0.4985314],
              [-0.969266,  1.8760108,  0.041556 ],
              [0.0556434, -0.2040259,  1.0572252]]

colorsRGB = applyMatrix(sRGBMatrix, colorsXYZ)
```

psychopy.tools.mathtools.posOriToMatrix

psychopy.tools.mathtools.**posOriToMatrix** (*pos, ori, out=None, dtype=None*)

Convert a rigid body pose to a 4x4 transformation matrix.

A pose is represented by a position coordinate *pos* and orientation quaternion *ori*.

Parameters

- **pos** (*ndarray, tuple, or list of float*) – Position vector [x, y, z].
- **ori** (*tuple, list or ndarray of float*) – Orientation quaternion in form [x, y, z, w] where w is real and x, y, z are imaginary components.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns 4x4 transformation matrix.

Return type ndarray

Collisions

Tools for determining whether a vector intersects a solid or bounding volume.

<code>fitBBox(points[, dtype])</code>	Fit an axis-aligned bounding box around points.
<code>computeBBoxCorners(extents[, dtype])</code>	Get the corners of an axis-aligned bounding box.
<code>intersectRayPlane(rayOrig, rayDir, ...[, dtype])</code>	Get the point which a ray intersects a plane.
<code>intersectRaySphere(rayOrig, rayDir[, ...])</code>	Calculate the points which a ray/line intersects a sphere (if any).
<code>intersectRayAABB(rayOrig, rayDir, ...[, dtype])</code>	Find the point a ray intersects an axis-aligned bounding box (AABB).
<code>intersectRayOBB(rayOrig, rayDir, ...[, dtype])</code>	Find the point a ray intersects an oriented bounding box (OBB).
<code>intersectRayTriangle(rayOrig, rayDir, tri[, ...])</code>	Get the intersection of a ray and triangle(s).

psychopy.tools.mathtools.fitBBox

`psychopy.tools.mathtools.fitBBox` (*points*, *dtype=None*)

Fit an axis-aligned bounding box around points.

This computes the minimum and maximum extents for a bounding box to completely enclose *points*. Keep in mind the the output in bounds are axis-aligned and may not optimally fits the points (i.e. fits the points with the minimum required volume). However, this should work well enough for applications such as visibility testing (see `~psychopy.tools.viewtools.volumeVisible` for more information..

Parameters

- **points** (*array_like*) – Nx3 or Nx4 array of points to fit the bounding box to.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Extents (mins, maxs) as a 2x3 array.

Return type ndarray

See also:

`computeBBoxCorners` () Convert bounding box extents to corners.

psychopy.tools.mathtools.computeBBoxCorners

`psychopy.tools.mathtools.computeBBoxCorners` (*extents*, *dtype=None*)

Get the corners of an axis-aligned bounding box.

Parameters

- **extents** (*array_like*) – 2x3 array indicating the minimum and maximum extents of the bounding box.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns 8x4 array of points defining the corners of the bounding box.

Return type ndarray

Examples

Compute the corner points of a bounding box:

```
minExtent = [-1, -1, -1]
maxExtent = [1, 1, 1]
corners = computeBBoxCorners([minExtent, maxExtent])

# [[ 1.  1.  1.  1.]
# [-1.  1.  1.  1.]
# [ 1. -1.  1.  1.]
# [-1. -1.  1.  1.]
# [ 1.  1. -1.  1.]
# [-1.  1. -1.  1.]
```

(continues on next page)

(continued from previous page)

```
# [ 1. -1. -1.  1.]
# [-1. -1. -1.  1.]
```

psychoPy.tools.mathtools.intersectRayPlane

psychoPy.tools.mathtools.**intersectRayPlane**(*rayOrig*, *rayDir*, *planeOrig*, *planeNormal*,
dtype=None)

Get the point which a ray intersects a plane.

Parameters

- **rayOrig** (*array_like*) – Origin of the line in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the line [x, y, z].
- **planeOrig** (*array_like*) – Origin of the plane to test [x, y, z].
- **planeNormal** (*array_like*) – Normal vector of the plane [x, y, z].
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Position (*ndarray*) in space which the line intersects the plane and the distance the intersect occurs from the origin (*float*). *None* is returned if the line does not intersect the plane at a single point or at all.

Return type *tuple* or *None*

Examples

Find the point in the scene a ray intersects the plane:

```
# plane information
planeOrigin = [0, 0, 0]
planeNormal = [0, 0, 1]
planeUpAxis = perp([0, 1, 0], planeNormal)

# ray
rayDir = [0, 0, -1]
rayOrigin = [0, 0, 5]

# get the intersect and distance in 3D world space
pnt, dist = intersectRayPlane(rayOrigin, rayDir, planeOrigin, planeNormal)
```

psychoPy.tools.mathtools.intersectRaySphere

psychoPy.tools.mathtools.**intersectRaySphere**(*rayOrig*, *rayDir*, *sphereOrig*=0.0, 0.0, 0.0,
sphereRadius=1.0, *dtype=None*)

Calculate the points which a ray/line intersects a sphere (if any).

Get the 3D coordinate of the point which the ray intersects the sphere and the distance to the point from *orig*. The nearest point is returned if the line intersects the sphere at multiple locations. All coordinates should be in world/scene units.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.
- **sphereOrig** (*array_like*) – Origin of the sphere to test [x, y, z].
- **sphereRadius** (*float*) – Sphere radius to test in scene units.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Coordinate in world space of the intersection and distance in scene units from *orig*. Returns *None* if there is no intersection.

Return type `tuple`

psychopy.tools.mathtools.intersectRayAABB

`psychopy.tools.mathtools.intersectRayAABB` (*rayOrig, rayDir, boundsOffset, boundsExtents, dtype=None*)

Find the point a ray intersects an axis-aligned bounding box (AABB).

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.
- **boundsOffset** (*array_like*) – Offset of the bounding box in the scene [x, y, z].
- **boundsExtents** (*array_like*) – Minimum and maximum extents of the bounding box.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type `tuple`

Examples

Get the point on an axis-aligned bounding box that the cursor is over and place a 3D stimulus there. The eye location is defined by *RigidBodyPose* object *camera*:

```
# get the mouse position on-screen
mx, my = mouse.getPos()

# find the point which the ray intersects on the box
result = intersectRayAABB(
    camera.pos,
    camera.transformNormal(win.coordToRay((mx, my))),
    myStim.pos,
    myStim.thePose.bounds.extents)

# if the ray intersects, set the position of the cursor object to it
if result is not None:
```

(continues on next page)

(continued from previous page)

```
cursorModel.thePose.pos = result[0]
cursorModel.draw() # don't draw anything if there is no intersect
```

Note that if the model is rotated, the bounding box may not be aligned anymore with the axes. Use *intersectRayOBB* if your model rotates.

psychopy.tools.mathtools.intersectRayOBB

`psychopy.tools.mathtools.intersectRayOBB` (*rayOrig*, *rayDir*, *modelMatrix*, *boundsExtents*, *dtype=None*)

Find the point a ray intersects an oriented bounding box (OBB).

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.
- **modelMatrix** (*array_like*) – 4x4 model matrix of the object and bounding box.
- **boundsExtents** (*array_like*) – Minimum and maximum extents of the bounding box.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be 'float32' or 'float64'. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, 'float64' is used by default.

Returns Coordinate in world space of the intersection and distance in scene units from *rayOrig*. Returns *None* if there is no intersection.

Return type tuple

Examples

Get the point on an oriented bounding box that the cursor is over and place a 3D stimulus there. The eye location is defined by *RigidBodyPose* object *camera*:

```
# get the mouse position on-screen
mx, my = mouse.getPos()

# find the point which the ray intersects on the box
result = intersectRayOBB(
    camera.pos,
    camera.transformNormal(win.coordToRay((mx, my))),
    myStim.thePose.getModelMatrix(),
    myStim.thePose.bounds.extents)

# if the ray intersects, set the position of the cursor object to it
if result is not None:
    cursorModel.thePose.pos = result[0]
    cursorModel.draw() # don't draw anything if there is no intersect
```

psychopy.tools.mathtools.intersectRayTriangle

`psychopy.tools.mathtools.intersectRayTriangle` (*rayOrig*, *rayDir*, *tri*, *dtype=None*)

Get the intersection of a ray and triangle(s).

This function can be used to achieve ‘pixel-perfect’ ray picking/casting on meshes defined with triangles. However, high-poly meshes may lead to performance issues.

Parameters

- **rayOrig** (*array_like*) – Origin of the ray in space [x, y, z].
- **rayDir** (*array_like*) – Direction vector of the ray [x, y, z], should be normalized.
- **tri** (*array_like*) – Triangle vertices as 2D (3x3) array [p0, p1, p2] where each vertex is a length 3 array [vx, xy, vz]. The input array can be 3D (Nx3x3) to specify multiple triangles.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Coordinate in world space of the intersection, distance in scene units from *rayOrig*, and the barycentric coordinates on the triangle [x, y]. Returns *None* if there is no intersection.

Return type tuple

Distortion

Functions for generating barrel/pincushion distortion meshes to correct image distortion. Such distortion is usually introduced by lenses in the optical path between the viewer and the display.

<code>lensCorrection(xys[, coefK, distCenter, ...])</code>	Lens correction (or distortion) using the division model with even polynomial terms.
<code>lensCorrectionSpherical(xys[, coefK, ...])</code>	Simple lens correction.

psychopy.tools.mathtools.lensCorrection

`psychopy.tools.mathtools.lensCorrection` (*xys*, *coefK=1.0*, *distCenter=0.0, 0.0*, *out=None*, *dtype=None*)

Lens correction (or distortion) using the division model with even polynomial terms.

Calculate new vertex positions or texture coordinates to apply radial warping, such as ‘pincushion’ and ‘barrel’ distortion. This is to compensate for optical distortion introduced by lenses placed in the optical path of the viewer and the display (such as in an HMD).

See references[1]_ for implementation details.

Parameters

- **xys** (*array_like*) – Nx2 list of vertex positions or texture coordinates to distort. Works correctly only if input values range between -1.0 and 1.0.
- **coefK** (*array_like or float*) – Distortion coefficients *K_n*. Specifying multiple values will add more polynomial terms to the distortion formula. Positive values will produce ‘barrel’ distortion, whereas negative will produce ‘pincushion’ distortion. In most cases, two or three coefficients are adequate, depending on the degree of distortion.

- **distCenter** (*array_like, optional*) – X and Y coordinate of the distortion center (eg. (0.2, -0.4)).
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Array of distorted vertices.

Return type ndarray

Notes

- At this time tangential distortion (i.e. due to a slant in the display) cannot be corrected for.

References

Examples

Creating a lens correction mesh with barrel distortion (eg. for HMDs):

```
vertices, textureCoords, normals, faces = gltools.createMeshGrid(
    subdiv=11, tessMode='center')

# recompute vertex positions
vertices[:, :2] = mt.lensCorrection(vertices[:, :2], coefK=(5., 5.))
```

psychoPy.tools.mathtools.lensCorrectionSpherical

`psychoPy.tools.mathtools.lensCorrectionSpherical` (*xy*, *coefK=1.0*, *aspect=1.0*, *out=None*, *dtype=None*)

Simple lens correction.

Lens correction for a spherical lenses with distortion centered at the middle of the display. See references[1]_ for implementation details.

Parameters

- **xy** (*array_like*) – Nx2 list of vertex positions or texture coordinates to distort. Assumes the output will be rendered to normalized device coordinates where points range from -1.0 to 1.0.
- **coefK** (*float*) – Distortion coefficient. Use positive numbers for pincushion distortion and negative for barrel distortion.
- **aspect** (*float*) – Aspect ratio of the target window or buffer (width / height).
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for computations can either be ‘float32’ or ‘float64’. If *out* is specified, the data type of *out* is used and this argument is ignored. If *out* is not provided, ‘float64’ is used by default.

Returns Array of distorted vertices.

Return type ndarray

References

Examples

Creating a lens correction mesh with barrel distortion (eg. for HMDs):

```
vertices, textureCoords, normals, faces = gltools.createMeshGrid(
    subdiv=11, tessMode='center')

# recompute vertex positions
vertices[:, :2] = mt.lensCorrection2(vertices[:, :2], coefK=2.0)
```

Miscellaneous

Miscellaneous and helper functions.

<code>zeroFix(a[, inplace, threshold])</code>	Fix zeros in an array.
---	------------------------

psychoPy.tools.mathtools.zeroFix

`psychoPy.tools.mathtools.zeroFix(a, inplace=False, threshold=None)`

Fix zeros in an array.

This function truncates very small numbers in an array to zero and removes any negative zeros.

Parameters

- **a** (*ndarray*) – Input array, must be a Numpy array.
- **inplace** (*bool*) – Fix an array inplace. If *True*, the input array will be modified, otherwise a new array will be returned with same *dtype* and shape with the fixed values.
- **threshold** (*float or None*) – Threshold for truncation. If *None*, the machine epsilon value for the input array *dtype* will be used. You can specify a custom threshold as a float.

Returns Output array with zeros fixed.

Return type ndarray

Performance and Optimization

Most functions listed here are very fast, however they are optimized to work on arrays of values (vectorization). Calling functions repeatedly (for instance within a loop), should be avoided as the CPU overhead associated with each function call (not to mention the loop itself) can be considerable.

For example, one may want to normalize a bunch of randomly generated vectors by calling `normalize()` on each row:

```
v = np.random.uniform(-1.0, 1.0, (1000, 4)) # 1000 length 4 vectors
vn = np.zeros((1000, 4)) # place to write values

# don't do this!
```

(continues on next page)

(continued from previous page)

```
for i in range(1000):
    vn[i, :] = normalize(v[i, :])
```

The same operation is completed in considerably less time by passing the whole array to the function like so:

```
normalize(v, out=vn) # very fast!
vn = normalize(v) # also fast if `out` is not provided
```

Specifying an output array to *out* will improve performance by reducing overhead associated with allocating memory to store the result (functions do this automatically if *out* is not provided). However, *out* should only be provided if the output array is reused multiple times. Furthermore, the function still returns a value if *out* is provided, but the returned value is a reference to *out*, not a copy of it. If *out* is not provided, the function will return the result with a freshly allocated array.

Data Types

Sub-routines used by the functions here will perform arithmetic using 64-bit floating-point precision unless otherwise specified via the *dtype* argument. This functionality is helpful in certain applications where input and output arrays demand a specific type (eg. when working with data passed to and from OpenGL functions).

If a *dtype* is specified, input arguments will be coerced to match that type and all floating-point arithmetic will use the precision of the type. If input arrays have the same type as *dtype*, they will automatically pass-through without being recast as a different type. As a performance consideration, all input arguments should have matching types and *dtype* set accordingly.

Most functions have an *out* argument, where one can specify an array to write values to. The value of *dtype* is ignored if *out* is provided, and all input arrays will be converted to match the *dtype* of *out* (if not already). This ensures that the type of the destination array is used for all arithmetic.

9.7.7 psychopy.tools.monitorunittools

Functions and classes related to unit conversion respective to a particular monitor

<code>convertToPix(vertices, pos, units, win)</code>	Takes vertices and position, combines and converts to pixels from any unit
<code>cm2deg(cm, monitor[, correctFlat])</code>	Convert size in cm to size in degrees for a given Monitor object
<code>cm2pix(cm, monitor)</code>	Convert size in cm to size in pixels for a given Monitor object.
<code>deg2cm(degrees, monitor[, correctFlat])</code>	Convert size in degrees to size in pixels for a given Monitor object.
<code>deg2pix(degrees, monitor[, correctFlat])</code>	Convert size in degrees to size in pixels for a given Monitor object
<code>pix2cm(pixels, monitor)</code>	Convert size in pixels to size in cm for a given Monitor object
<code>pix2deg(pixels, monitor[, correctFlat])</code>	Convert size in pixels to size in degrees for a given Monitor object

Function details

`psychoPy.tools.monitorunittools.convertToPix` (*vertices, pos, units, win*)

Takes vertices and position, combines and converts to pixels from any unit

The reason that *pos* and *vertices* are provided separately is that it allows the conversion from deg to apply flat-screen correction to each separately.

The reason that these use function args rather than relying on `self.pos` is that some stimuli use other terms (e.g. `ElementArrayStim` uses `fieldPos`).

`psychoPy.tools.monitorunittools.cm2deg` (*cm, monitor, correctFlat=False*)

Convert size in cm to size in degrees for a given Monitor object

`psychoPy.tools.monitorunittools.cm2pix` (*cm, monitor*)

Convert size in cm to size in pixels for a given Monitor object.

`psychoPy.tools.monitorunittools.deg2cm` (*degrees, monitor, correctFlat=False*)

Convert size in degrees to size in pixels for a given Monitor object.

If *correctFlat* == *False* then the screen will be treated as if all points are equal distance from the eye. This means that each “degree” will be the same size irrespective of its position.

If *correctFlat* == *True* then the *degrees* argument must be an Nx2 matrix for X and Y values (the two cannot be calculated separately in this case).

With *correctFlat* == *True* the positions may look strange because more eccentric vertices will be spaced further apart.

`psychoPy.tools.monitorunittools.deg2pix` (*degrees, monitor, correctFlat=False*)

Convert size in degrees to size in pixels for a given Monitor object

`psychoPy.tools.monitorunittools.pix2cm` (*pixels, monitor*)

Convert size in pixels to size in cm for a given Monitor object

`psychoPy.tools.monitorunittools.pix2deg` (*pixels, monitor, correctFlat=False*)

Convert size in pixels to size in degrees for a given Monitor object

9.7.8 `psychoPy.tools.plottools`

Functions and classes related to plotting

`psychoPy.tools.plottools.plotFrameIntervals` (*intervals*)

Plot a histogram of the frame intervals.

Where *intervals* is either a filename to a file, saved by `Window.saveFrameIntervals`, or simply a list (or array) of frame intervals

9.7.9 `psychoPy.tools.rifttools`

Various tools for working with the *Rift* class. The documentation for classes in on this page originate from PsychXR and may make references to functions and objects not included with .

Overview

Classes

These classes are included with PsychXR to use with the LibOVR interface. They can be accessed from this module to avoid needing to explicitly import PsychXR. If PsychXR is not available on the system, these classes will have values *None*.

LibOVRPose

LibOVRPoseState

LibOVRHapticsBuffer

LibOVRBounds

Functions

These functions can be called without first starting a VR session (initializing a *Rift* instance) to check if the drivers/services are running on this computer or if an HMD is connected.

isHmdConnected([timeout])

Check if an HMD is connected.

isOculusServiceRunning([timeout])

Check if the Oculus(tm) service is currently running.

Details

`psychopy.tools.rifttools.LibOVRPose`

`psychopy.tools.rifttools.LibOVRPoseState`

`psychopy.tools.rifttools.LibOVRBounds`

`psychopy.tools.rifttools.LibOVRHapticsBuffer`

`psychopy.tools.rifttools.isHmdConnected` (*timeout=0*)

Check if an HMD is connected.

Parameters *timeout* (*int*) – Timeout in milliseconds.

Returns *True* if an HMD is connected.

Return type *bool*

`psychopy.tools.rifttools.isOculusServiceRunning` (*timeout=0*)

Check if the Oculus(tm) service is currently running.

Parameters *timeout* (*int*) – Timeout in milliseconds.

Returns *True* if the service is loaded and running.

Return type *bool*

9.7.10 `psychopy.tools.typetools`

Functions and classes related to variable type conversion

`psychopy.tools.typetools.float_uint8` (*inarray*)

Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint8s ranging 0:255

```
>>> float_uint8(-1)
0
>>> float_uint8(0)
128
```

`psychopy.tools.typetools.uint8_float` (*inarray*)

Converts arrays, lists, tuples and UINTs ranging 0:255 into an array of floats ranging -1:1

```
>>> uint8_float(0)
-1.0
>>> uint8_float(128)
0.0
```

`psychopy.tools.typetools.float_uint16` (*inarray*)

Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint16s ranging 0:2¹⁶

```
>>> float_uint16(-1)
0
>>> float_uint16(0)
32768
```

9.7.11 `psychopy.tools.unittools`

Functions and classes related to unit conversion

`psychopy.tools.unittools.radians` (*x*, *l*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*, [*signature*, *extobj*])

Convert angles from degrees to radians.

Parameters

- **x** (*array_like*) – Input array in degrees.
- **out** (*ndarray, None, or tuple of ndarray and None, optional*) – A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** (*array_like, optional*) – This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is False will remain uninitialized.
- ****kwargs** – For other keyword-only arguments, see the [ufunc docs](#).

Returns *y* – The corresponding radian values. This is a scalar if *x* is a scalar.

Return type ndarray

See also:

`deg2rad()` equivalent function

Examples

Convert a degree array to radians

```
>>> deg = np.arange(12.) * 30.
>>> np.radians(deg)
array([ 0.          ,  0.52359878,  1.04719755,  1.57079633,  2.0943951 ,
        2.61799388,  3.14159265,  3.66519143,  4.1887902 ,  4.71238898,
        5.23598776,  5.75958653])
```

```
>>> out = np.zeros((deg.shape))
>>> ret = np.radians(deg, out)
>>> ret is out
True
```

`psychopy.tools.unittools.degrees` (*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True* [, *signature*, *extobj*])

Convert angles from radians to degrees.

Parameters

- **x** (*array_like*) – Input array in radians.
- **out** (*ndarray, None, or tuple of ndarray and None, optional*) – A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or `None`, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.
- **where** (*array_like, optional*) – This condition is broadcast over the input. At locations where the condition is `True`, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default *out=None*, locations within it where the condition is `False` will remain uninitialized.
- ****kwargs** – For other keyword-only arguments, see the [ufunc docs](#).

Returns *y* – The corresponding degree values; if *out* was supplied this is a reference to it. This is a scalar if *x* is a scalar.

Return type ndarray of floats

See also:

`rad2deg()` equivalent function

Examples

Convert a radian array to degrees

```
>>> rad = np.arange(12.)*np.pi/6
>>> np.degrees(rad)
array([ 0.,  30.,  60.,  90., 120., 150., 180., 210., 240.,
        270., 300., 330.] )
```

```

>>> out = np.zeros((rad.shape))
>>> r = np.degrees(rad, out)
>>> np.all(r == out)
True
    
```

9.7.12 psychopy.tools.viewtools

Math functions for working with view transformations and performing visibility testing (see also *mathtools*).

Tools for working with view projections for 2- and 3-D rendering.

Overview

<code>visualAngle(size, distance[, degrees, out, ...])</code>	Get the visual angle for an object of <i>size</i> at <i>distance</i> .
<code>computeFrustum(scrWidth, scrAspect, scrDist)</code>	Calculate frustum parameters.
<code>computeFrustumFOV(scrFOV, scrAspect, scrDist)</code>	Compute a frustum for a given field-of-view (FOV).
<code>projectFrustum(frustum, dist[, dtype])</code>	Project a frustum on a fronto-parallel plane and get the width and height of the required drawing area.
<code>projectFrustumToPlane(frustum, planeOrig[, ...])</code>	Project a frustum on a fronto-parallel plane and get the coordinates of the corners in physical space.
<code>generalizedPerspectiveProjection(...[, ...])</code>	Generalized derivation of projection and view matrices based on the physical configuration of the display system.
<code>orthoProjectionMatrix(left, right, bottom, top)</code>	Compute an orthographic projection matrix with provided frustum parameters.
<code>perspectiveProjectionMatrix(left, right, ...)</code>	Compute an perspective projection matrix with provided frustum parameters.
<code>lookAt(eyePos, centerPos[, upVec, out, dtype])</code>	Create a transformation matrix to orient a view towards some point.
<code>pointToNdc(wcsPos, viewMatrix, projectionMatrix)</code>	Map the position of a point in world space to normalized device coordinates/space.
<code>cursorToRay(cursorX, cursorY, winSize, ...)</code>	Convert a 2D mouse coordinate to a 3D ray.
<code>visible(points, mvp[, mode, dtype])</code>	Test if points are visible.
<code>visibleBBox(extents, mvp[, dtype])</code>	Check if a bounding box is visible.

Details

`psychopy.tools.viewtools.visualAngle` (*size, distance, degrees=True, out=None, dtype=None*)

Get the visual angle for an object of *size* at *distance*. Object is assumed to be fronto-parallel with the viewer.

This function supports vector inputs. Values for *size* and *distance* can be arrays or single values. If both inputs are arrays, they must have the same size.

Parameters

- **size** (*float* or *array_like*) – Size of the object in meters.
- **distance** (*float* or *array_like*) – Distance to the object in meters.
- **degrees** (*bool*) – Return result in degrees, if *False* result will be in radians.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.

- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns Visual angle.

Return type float

Examples

Calculating the visual angle (vertical FOV) of a monitor screen:

```
monDist = 0.5 # monitor distance, 50cm
monHeight = 0.45 # monitor height, 45cm

vertFOV = visualAngle(monHeight, monDist)
```

Compute visual angle at multiple distances for objects with the same size:

```
va = visualAngle(0.20, [1.0, 2.0, 3.0]) # returns
# [11.42118627 5.72481045 3.81830487]
```

`psychopy.tools.viewtools.computeFrustum` (*scrWidth, scrAspect, scrDist, convergeOffset=0.0, eyeOffset=0.0, nearClip=0.01, farClip=100.0, dtype=None*)

Calculate frustum parameters. If an eye offset is provided, an asymmetric frustum is returned which can be used for stereoscopic rendering.

Parameters

- **scrWidth** (*float*) – The display’s width in meters.
- **scrAspect** (*float*) – Aspect ratio of the display (width / height).
- **scrDist** (*float*) – Distance to the screen from the view in meters. Measured from the center of their eyes.
- **convergeOffset** (*float*) – Offset of the convergence plane from the screen. Objects falling on this plane will have zero disparity. For best results, the convergence plane should be set to the same distance as the screen (0.0 by default).
- **eyeOffset** (*float*) – Half the inter-ocular separation (i.e. the horizontal distance between the nose and center of the pupil) in meters. If *eyeOffset* is 0.0, a symmetric frustum is returned.
- **nearClip** (*float*) – Distance to the near clipping plane in meters from the viewer. Should be at least less than *scrDist*.
- **farClip** (*float*) – Distance to the far clipping plane from the viewer in meters. Must be >*nearClip*.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns Array of frustum parameters. Can be directly passed to `glFrustum` (e.g. `glFrustum(*f)`).

Return type ndarray

Notes

- The view point must be transformed for objects to appear correctly. Offsets in the X-direction must be applied +/- eyeOffset to account for inter-ocular separation. A transformation in the Z-direction must be applied to account for screen distance. These offsets **MUST** be applied to the GL_MODELVIEW matrix, not the GL_PROJECTION matrix! Doing so may break lighting calculations.

Examples

Creating a frustum and setting a window's projection matrix:

```
scrWidth = 0.5 # screen width in meters
scrAspect = win.size[0] / win.size[1]
scrDist = win.scrDistCM * 100.0 # monitor setting, can be anything
frustum = viewtools.computeFrustum(scrWidth, scrAspect, scrDist)
```

Accessing frustum parameters:

```
left, right, bottom, top, nearVal, farVal = frustum
# ... or ...
left = frustum.left
```

Off-axis frustums for stereo rendering:

```
# compute view matrix for each eye, these value usually don't change
eyeOffset = (-0.035, 0.035) # +/- IOD / 2.0
scrDist = 0.50 # 50cm
scrWidth = 0.53 # 53cm
scrAspect = 1.778
leftFrustum = viewtools.computeFrustum(
    scrWidth, scrAspect, scrDist, eyeOffset[0])
rightFrustum = viewtools.computeFrustum(
    scrWidth, scrAspect, scrDist, eyeOffset[1])
# make sure your view matrix accounts for the screen distance and eye
# offsets!
```

Using computed view frustums with a window:

```
win.projectionMatrix = viewtools.perspectiveProjectionMatrix(*frustum)
# generate a view matrix looking ahead with correct viewing distance,
# origin is at the center of the screen. Assumes eye is centered with
# the screen.
eyePos = [0.0, 0.0, scrDist]
screenPos = [0.0, 0.0, 0.0] # look at screen center
eyeUp = [0.0, 1.0, 0.0]
win.viewMatrix = viewtools.lookAt(eyePos, screenPos, eyeUp)
win.applyViewTransform() # call before drawing
```

`psychopy.tools.viewtools.computeFrustumFOV(scrFOV, scrAspect, scrDist, convergeOffset=0.0, eyeOffset=0.0, nearClip=0.01, farClip=100.0, dtype=None)`

Compute a frustum for a given field-of-view (FOV).

Similar to `computeFrustum`, but computes a frustum based on FOV rather than screen dimensions.

Parameters

- **scrFOV** (*float*) – Vertical FOV in degrees (fovY).

- **scrAspect** (*float*) – Aspect between the horizontal and vertical FOV (ie. fovX / fovY).
- **scrDist** (*float*) – Distance to the screen from the view in meters. Measured from the center of the viewer’s eye(s).
- **convergeOffset** (*float*) – Offset of the convergence plane from the screen. Objects falling on this plane will have zero disparity. For best results, the convergence plane should be set to the same distance as the screen (0.0 by default).
- **eyeOffset** (*float*) – Half the inter-ocular separation (i.e. the horizontal distance between the nose and center of the pupil) in meters. If eyeOffset is 0.0, a symmetric frustum is returned.
- **nearClip** (*float*) – Distance to the near clipping plane in meters from the viewer. Should be at least less than *scrDist*. Never should be 0.
- **farClip** (*float*) – Distance to the far clipping plane from the viewer in meters. Must be >nearClip.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Examples

Equivalent to *gluPerspective*:

```
frustum = computeFrustumFOV(45.0, 1.0, 0.5)
projectionMatrix = perspectiveProjectionMatrix(*frustum)
```

`psychopy.tools.viewtools.projectFrustum` (*frustum, dist, dtype=None*)

Project a frustum on a fronto-parallel plane and get the width and height of the required drawing area.

This function can be used to determine the size of the drawing area required for a given frustum on a screen. This is useful for cases where the observer is viewing the screen through a physical aperture that limits the FOV to a sub-region of the display. You must convert the size in meters to units of your screen and apply any offsets.

Parameters

- **frustum** (*array_like*) – Frustum parameters (left, right, bottom, top, near, far), you can exclude *far* since it is not used in this calculation. However, the function will still succeed if given.
- **dist** (*float*) – Distance to project points to in meters.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns Width and height (w, h) of the area intersected by the given frustum at *dist*.

Return type ndarray

Examples

Compute the viewport required to draw in the area where the frustum intersects the screen:

```
# needed information
scrWidthM = 0.52
scrDistM = 0.72
scrWidthPIX = 1920
scrHeightPIX = 1080
scrAspect = scrWidthPIX / float(scrHeightPIX)
pixPerMeter = scrWidthPIX / scrWidthM

# Compute a frustum for 20 degree vertical FOV at distance of the
# screen.
frustum = computeFrustumFOV(20., scrAspect, scrDistM)

# get the dimensions of the frustum
w, h = projectFrustum(frustum, scrDistM) * pixPerMeter

# get the origin of the viewport, relative to center of screen.
x = (scrWidthPIX - w) / 2.
y = (scrHeightPIX - h) / 2.

# if there is an eye offset ...
# x = (scrWidthPIX - w + eyeOffsetM * pixPerMeter) / 2.

# viewport rectangle
rect = np.asarray((x, y, w, h), dtype=int)
```

You can then set the viewport/scissor rectangle of the buffer to restrict drawing to *rect*.

`psychoPy.tools.viewtools.projectFrustumToPlane` (*frustum*, *planeOrig*, *dtype=None*)
Project a frustum on a fronto-parallel plane and get the coordinates of the corners in physical space.

Parameters

- **frustum** (*array_like*) – Frustum parameters (left, right, bottom, top, near, far), you can exclude *far* since it is not used in this calculation. However, the function will still succeed if given.
- **planeOrig** (*float*) – Distance of plane to project points on in meters.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns 4x3 array of coordinates in the physical reference frame with origin at the eye.

Return type ndarray

`psychoPy.tools.viewtools.generalizedPerspectiveProjection` (*posBottomLeft*, *posBottomRight*, *posTopLeft*, *eyePos*, *nearClip=0.01*, *farClip=100.0*, *dtype=None*)

Generalized derivation of projection and view matrices based on the physical configuration of the display system.

This implementation is based on Robert Kooima’s ‘Generalized Perspective Projection’ method¹.

Parameters

¹ Kooima, R. (2009). Generalized perspective projection. J. Sch. Electron. Eng. Comput. Sci.

- **posBottomLeft** (*list of float or ndarray*) – Bottom-left 3D coordinate of the screen in meters.
- **posBottomRight** (*list of float or ndarray*) – Bottom-right 3D coordinate of the screen in meters.
- **posTopLeft** (*list of float or ndarray*) – Top-left 3D coordinate of the screen in meters.
- **eyePos** (*list of float or ndarray*) – Coordinate of the eye in meters.
- **nearClip** (*float*) – Near clipping plane distance from viewer in meters.
- **farClip** (*float*) – Far clipping plane distance from viewer in meters.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns The 4x4 projection and view matrix.

Return type `tuple`

See also:

`computeFrustum()` Compute frustum parameters.

Notes

- The resulting projection frustums are off-axis relative to the center of the display.
- The returned matrices are row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

References

Examples

Computing a projection and view matrices for a window:

```
projMatrix, viewMatrix = viewtools.generalizedPerspectiveProjection(
    posBottomLeft, posBottomRight, posTopLeft, eyePos)
# set the window matrices
win.projectionMatrix = projMatrix
win.viewMatrix = viewMatrix
# before rendering
win.applyEyeTransform()
```

Stereo-pair rendering example from Kooima (2009):

```
# configuration of screen and eyes
posBottomLeft = [-1.5, -0.75, -18.0]
posBottomRight = [1.5, -0.75, -18.0]
posTopLeft = [-1.5, 0.75, -18.0]
posLeftEye = [-1.25, 0.0, 0.0]
posRightEye = [1.25, 0.0, 0.0]
# create projection and view matrices
leftProjMatrix, leftViewMatrix = generalizedPerspectiveProjection(
```

(continues on next page)

(continued from previous page)

```
posBottomLeft, posBottomRight, posTopLeft, posLeftEye)
rightProjMatrix, rightViewMatrix = generalizedPerspectiveProjection(
    posBottomLeft, posBottomRight, posTopLeft, posRightEye)
```

`psychopy.tools.viewtools.orthoProjectionMatrix` (*left, right, bottom, top, nearClip=0.01, farClip=100.0, out=None, dtype=None*)

Compute an orthographic projection matrix with provided frustum parameters.

Parameters

- **left** (*float*) – Left clipping plane coordinate.
- **right** (*float*) – Right clipping plane coordinate.
- **bottom** (*float*) – Bottom clipping plane coordinate.
- **top** (*float*) – Top clipping plane coordinate.
- **nearClip** (*float*) – Near clipping plane distance from viewer.
- **farClip** (*float*) – Far clipping plane distance from viewer.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns 4x4 projection matrix

Return type ndarray

See also:

[*perspectiveProjectionMatrix\(\)*](#) Compute a perspective projection matrix.

Notes

- The returned matrix is row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

`psychopy.tools.viewtools.perspectiveProjectionMatrix` (*left, right, bottom, top, nearClip=0.01, farClip=100.0, out=None, dtype=None*)

Compute an perspective projection matrix with provided frustum parameters. The frustum can be asymmetric.

Parameters

- **left** (*float*) – Left clipping plane coordinate.
- **right** (*float*) – Right clipping plane coordinate.
- **bottom** (*float*) – Bottom clipping plane coordinate.
- **top** (*float*) – Top clipping plane coordinate.
- **nearClip** (*float*) – Near clipping plane distance from viewer.
- **farClip** (*float*) – Far clipping plane distance from viewer.

- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns 4x4 projection matrix

Return type ndarray

See also:

`orthoProjectionMatrix()` Compute an orthographic projection matrix.

Notes

- The returned matrix is row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

`psychopy.tools.viewtools.lookAt` (*eyePos, centerPos, upVec=0.0, 1.0, 0.0, out=None, dtype=None*)

Create a transformation matrix to orient a view towards some point. Based on the same algorithm as ‘gluLookAt’. This does not generate a projection matrix, but rather the matrix to transform the observer’s view in the scene.

For more information see: <https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/gluLookAt.xml>

Parameters

- **eyePos** (*list of float or ndarray*) – Eye position in the scene.
- **centerPos** (*list of float or ndarray*) – Position of the object center in the scene.
- **upVec** (*list of float or ndarray, optional*) – Vector defining the up vector. Default is +Y is up.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns 4x4 view matrix

Return type ndarray

Notes

- The returned matrix is row-major. Values are floats with 32-bits of precision stored as a contiguous (C-order) array.

`psychopy.tools.viewtools.pointToNdc` (*wcsPos, viewMatrix, projectionMatrix, out=None, dtype=None*)

Map the position of a point in world space to normalized device coordinates/space.

Parameters

- **wcsPos** (*tuple, list or ndarray*) – Nx3 position vector(s) (xyz) in world space coordinates.
- **viewMatrix** (*ndarray*) – 4x4 view matrix.
- **projectionMatrix** (*ndarray*) – 4x4 projection matrix.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns 3x1 vector of normalized device coordinates with type ‘float32’

Return type ndarray

Notes

- The point is not visible, falling outside of the viewing frustum, if the returned coordinates fall outside of -1 and 1 along any dimension.
- In the rare instance the point falls directly on the eye in world space where the frustum converges to a point (singularity), the divisor will be zero during perspective division. To avoid this, the divisor is ‘bumped’ to 1e-5.
- This function assumes the display area is rectilinear. Any distortion or warping applied in normalized device or viewport space is not considered.

Examples

Determine if a point is visible:

```
point = (0.0, 0.0, 10.0) # behind the observer
ndc = pointToNdc(point, win.viewMatrix, win.projectionMatrix)
isVisible = not np.any((ndc > 1.0) | (ndc < -1.0))
```

Convert NDC to viewport (or pixel) coordinates:

```
scrRes = (1920, 1200)
point = (0.0, 0.0, -5.0) # forward -5.0 from eye
x, y, z = pointToNdc(point, win.viewMatrix, win.projectionMatrix)
pixelX = ((x + 1.0) / 2.0) * scrRes[0]
pixelY = ((y + 1.0) / 2.0) * scrRes[1]
# object at point will appear at (pixelX, pixelY)
```

`psychopy.tools.viewtools.cursorToRay(cursorX, cursorY, winSize, viewport, projectionMatrix, normalize=True, out=None, dtype=None)`

Convert a 2D mouse coordinate to a 3D ray.

Takes a 2D window/mouse coordinate and transforms it to a 3D direction vector from the viewpoint in eye space (vector origin is [0, 0, 0]). The center of the screen projects to vector [0, 0, -1].

Parameters

- **cursorX** (*float or int*) – Window coordinates. These need to be scaled if you are using a framebuffer that does not have 1:1 pixel mapping (i.e. retina display).

- **cursorY** (*float or int*) – Window coordinates. These need to be scaled if you are using a framebuffer that does not have 1:1 pixel mapping (i.e. retina display).
- **winSize** (*array_like*) – Size of the window client area [w, h].
- **viewport** (*array_like*) – Viewport rectangle [x, y, w, h] being used.
- **projectionMatrix** (*ndarray*) – 4x4 projection matrix being used.
- **normalize** (*bool*) – Normalize the resulting vector.
- **out** (*ndarray, optional*) – Optional output array. Must be same *shape* and *dtype* as the expected output if *out* was not specified.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns Direction vector (x, y, z).

Return type ndarray

Examples

Place a 3D stim at the mouse location 5.0 scene units (meters) away:

```
# define camera
camera = RigidBodyPose((-3.0, 5.0, 3.5))
camera.alignTo((0, 0, 0))

# in the render loop

dist = 5.0
mouseRay = vt.cursorToRay(x, y, win.size, win.viewport, win.projectionMatrix)
mouseRay *= dist # scale the vector

# set the sphere position by transforming vector to world space
sphere.thePose.pos = camera.transform(mouseRay)
```

`psychopy.tools.viewtools.visible` (*points,.mvp, mode='discrete', dtype=None*)

Test if points are visible.

This function is useful for visibility culling, where objects are only drawn if a portion of them are visible. This test can avoid costly drawing calls and OpenGL state changes if the object is not visible.

Parameters

- **points** (*array_like*) – Point(s) or bounding box to test. Input array must be Nx3 or Nx4, where each row is a point. It is recommended that the input be Nx4 since the *w* component will be appended if the input is Nx3 which adds overhead.
- **mvp** (*array_like*) – 4x4 MVP matrix.
- **mode** (*str*) – Test mode. If ‘discrete’, rows of *points* are treated as individual points. This function will return an array of boolean values with length equal to the number of rows in *points*, where the value at each index corresponds to the visibility test results for points at the matching row index of *points*. If ‘group’ a single boolean value is returned, which is *False* if all points fall to one side of the frustum.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns Test results. The type returned depends on *mode*.

Return type `bool` or `ndarray`

Examples

Visibility culling, only a draw line connecting two points if visible:

```
linePoints = [[-1.0, -1.0, -1.0, 1.0],
              [ 1.0,  1.0,  1.0, 1.0]]

mvp = np.matmul(win.projectionMatrix, win.viewMatrix)
if visible(linePoints, mvp, mode='group'):
    # drawing commands here ...
```

`psychopy.tools.viewtools.visibleBBBox` (*extents*, *mvp*, *dtype=None*)

Check if a bounding box is visible.

This function checks if a bonding box intersects a frustum defined by the current projection matrix, after being transformed by the model-view matrix.

Parameters

- **extents** (*array_like*) – Bounding box minimum and maximum extents as a 2x3 array. The first row if the minimum extents along each axis, and the second row the maximum extents (eg. `[[minX, minY, minZ], [maxX, maxY, maxZ]]`).
- **mvp** (*array_like*) – 4x4 MVP matrix.
- **dtype** (*dtype or str, optional*) – Data type for arrays, can either be ‘float32’ or ‘float64’. If *None* is specified, the data type is inferred by *out*. If *out* is not provided, the default is ‘float64’.

Returns Visibility test results.

Return type `ndarray` or `bool`

9.8 psychopy.app - the application suite

This module contains everything needed to run and manage the wxPython GUI application suite (e.g., Coder, Builder and Runner).

The functions presented here provide a simple interface to the application instance and any frames associated with it. This interface is intended for unit testing the GUI and for developers wishing to extend it.

9.9 Overview

<code>startApp([showSplash, testMode, safeMode])</code>	Start the PsychoPy GUI.
<code>quitApp()</code>	Quit the running PsychoPy application instance.
<code>isAppStarted()</code>	Check if the GUI portion of PsychoPy is running.
<code>getAppInstance()</code>	Get a reference to the <i>PsychoPyApp</i> object.
<code>getAppFrame(frameName)</code>	Get the reference to one of PsychoPy’s application frames.

9.10 Details

`psychopy.app.startApp(showSplash=True, testMode=False, safeMode=False)`

Start the PsychoPy GUI.

This function is idempotent, where additional calls after the app starts will have no effect unless `quitApp()` was previously called. After this function returns, you can get the handle to the created `PsychoPyApp` instance by calling `getAppInstance()` (returns `None` otherwise).

Errors raised during initialization due to unhandled exceptions with respect to the GUI application are usually fatal. You can examine ‘last_app_load.log’ inside the ‘psychopy3’ user directory (specified by preference ‘userPrefsDir’) to see the traceback. After startup, unhandled exceptions will appear in a special dialog box that shows the error traceback and provides some means to recover their work. Regular logging messages will appear in the log file or GUI. We use a separate error dialog here to delineate errors occurring in the user’s experiment scripts and those of the application itself.

Parameters

- **showSplash** (*bool*) – Show the splash screen on start.
- **testMode** (*bool*) – Must be *True* if creating an instance for unit testing.
- **safeMode** (*bool*) – Start PsychoPy in safe-mode. If *True*, the GUI application will launch with without plugins and will use a default a configuration (planned feature, not implemented yet).

`psychopy.app.quitApp()`

Quit the running PsychoPy application instance.

Will have no effect if `startApp()` has not been called previously.

`psychopy.app.isAppStarted()`

Check if the GUI portion of PsychoPy is running.

Returns *True* if the GUI is started else *False*.

Return type `bool`

`psychopy.app.getAppInstance()`

Get a reference to the `PsychoPyApp` object.

This function will return `None` if PsychoPy has been imported as a library or the app has not been fully realized.

Returns Handle to the application instance. Returns `None` if the app has not been started yet or the PsychoPy is being used without a GUI.

Return type `PsychoPyApp` or `None`

Examples

Get the coder frame (if any):

```
import psychopy.app as app
coder = app.getAppInstance().coder
```

`psychopy.app.getAppFrame(frameName)`

Get the reference to one of PsychoPy’s application frames. Returns `None` if the specified frame has not been fully realized yet or PsychoPy is not in GUI mode.

Parameters **frameName** (*str*) – Identifier for the frame to get a reference to. Valid names are ‘coder’, ‘builder’ or ‘runner’.

Returns Reference to the frame instance (i.e. *CoderFrame*, *BuilderFrame* or *RunnerFrame*). *None* is returned if the frame has not been created or the app is not running. May return a list if more than one window is opened.

Return type `object` or `None`

9.11 psychopy.colors - For working with colors

Classes and functions for working with colors.

9.11.1 Overview

<code>Color([color, space, contrast, conematrix])</code>	A class to store color details, knows what colour space it's in and can supply colours in any space.
<code>isValidColor(color[, space])</code>	Deprecated as of 2021.0
<code>hex2rgb255(hexColor)</code>	Deprecated as of 2021.0

9.11.2 Details

class `psychopy.colors.Color` (*color=None, space=None, contrast=None, conematrix=None*)

A class to store color details, knows what colour space it's in and can supply colours in any space.

Parameters

- **color** (*ArrayLike* or *None*) – Color values (coordinates). Value must be in a format applicable to the specified *space*.
- **space** (*str* or *None*) – Colorspace to interpret the value of *color* as being within.
- **contrast** (*int* or *float*) – Factor to modulate the contrast of the color.
- **conematrix** (*ArrayLike* or *None*) – Cone matrix for colorspaces which require it. Must be a 3x3 array.

property `alpha`

How opaque (1) or transparent (0) this color is. Synonymous with *opacity*.

property `copy()`

Return a duplicate of this colour

property `dkl`

Color value expressed as a DKL triplet.

property `dklCart`

Color value expressed as a cartesian DKL triplet.

property `dkla`

Color value expressed as a DKL triplet, with alpha value (0 to 1).

property `dklaCart`

Color value expressed as a cartesian DKL triplet, with alpha value (0 to 1).

property `hex`

Color value expressed as a hex string. Can be a '#' followed by 6 values from 0 to F (e.g. #F2545B).

property hsv

Color value expressed as an HSV triplet.

property hsva

Color value expressed as an HSV triplet, with alpha value (0 to 1).

property lms

Color value expressed as an LMS triplet.

property lmsa

Color value expressed as an LMS triplet, with alpha value (0 to 1).

property named

The name of this color, if it has one (*str*).

property opacity

How opaque (1) or transparent (0) this color is (*float*). Synonymous with *alpha*.

render (*space='rgb'*)

Apply contrast to the base color value and return the adjusted color value.

property rgb

Color value expressed as an RGB triplet from -1 to 1.

property rgb1

Color value expressed as an RGB triplet from 0 to 1.

property rgb255

Color value expressed as an RGB triplet from 0 to 255.

property rgba

Color value expressed as an RGB triplet from -1 to 1, with alpha values (0 to 1).

property rgba1

Color value expressed as an RGB triplet from 0 to 1, with alpha value (0 to 1).

property rgba255

Color value expressed as an RGB triplet from 0 to 255, with alpha value (0 to 1).

set (*color=None, space=None*)

Set the colour of this object - essentially the same as what happens on creation, but without having to initialise a new object.

property srgb

Color value expressed as an sRGB triplet

validate (*color, space=None*)

Check that a color value is valid in the given space, or all spaces if *space==None*.

`psychopy.colors.isValidColor` (*color, space='rgb'*)

Deprecated as of 2021.0

`psychopy.colors.hex2rgb255` (*hexColor*)

Deprecated as of 2021.0

Converts a hex color string (e.g. "#05ff66") into an rgb triplet ranging from 0:255

9.12 psychopy.data - functions for storing/saving/analysing data

Contents:

- *ExperimentHandler* - to combine multiple loops in one study
- *TrialHandler* - basic predefined trial matrix
- *TrialHandler2* - similar to TrialHandler but with ability to update mid-run
- *TrialHandlerExt* - similar to TrialHandler but with ability to run oddball designs
- *StairHandler* - for basic up-down (fixed step) staircases
- *QuestHandler* - for traditional QUEST algorithm
- *QuestPlusHandler* - for the updated QUEST+ algorithm (Watson, 2017)
- *PsiHandler* - the Psi staircase of Kontsevich & Tyler (1999)
- *MultiStairHandler* - a wrapper to combine interleaved staircases of any sort

Utility functions:

- *importConditions()* - to load a list of dicts from a csv/excel file
- *functionFromStaircase()* - to convert a staircase into its psychometric function
- *bootStraps()* - generate a set of bootstrap resamples from a dataset
- *getDateStr()* - provide a date string (in format suitable for filenames)

Curve Fitting:

- *FitWeibull*
- *FitLogistic*
- *FitNakaRushton*
- *FitCumNormal*

9.12.1 ExperimentHandler

```
class psychopy.data.ExperimentHandler (name="", version="", extraInfo=None, runtime-Info=None, originPath=None, savePickle=True, saveWideText=True, dataFileName="", autoLog=True, appendFiles=False)
```

A container class for keeping track of multiple loops/handlers

Useful for generating a single data file from an experiment with many different loops (e.g. interleaved staircases or loops within loops)

Usage `exp = data.ExperimentHandler(name="Face Preference",version='0.1.0')`

Parameters

name [a string or unicode] As a useful identifier later

version [usually a string (e.g. '1.1.0')] To keep track of which version of the experiment was run

extraInfo [a dictionary] Containing useful information about this run (e.g. {'participant':'jwp','gender':'m','orientation':90})

runtimeInfo [*psychopy.info.RunTimeInfo*] Containing information about the system as detected at runtime

originPath [string or unicode] The path and filename of the originating script/experiment If not provided this will be determined as the path of the calling script.

dataFileName [string] This is defined in advance and the file will be saved at any point that the handler is removed or discarded (unless `.abort()` had been called in advance). The handler will attempt to populate the file even in the event of a (not too serious) crash!

savePickle : True (default) or False

saveWideText : True (default) or False

autoLog : True (default) or False

getAllParamNames ()

Returns the attribute names of loop parameters (trialN etc) that the current set of loops contain, ready to build a wide-format data file.

getExtraInfo ()

Get the names and vals from the extraInfo dict (if it exists)

getLoopInfo (*loop*)

Returns the attribute names and values for the current trial of a particular loop. Does not return data inputs from the subject, only info relating to the trial execution.

abort ()

Inform the ExperimentHandler that the run was aborted.

Experiment handler will attempt automatically to save data (even in the event of a crash if possible). So if you quit your script early you may want to tell the Handler not to save out the data files for this run. This is the method that allows you to do that.

addData (*name, value*)

Add the data with a given name to the current experiment.

Typically the user does not need to use this function; if you added your data to the loop and had already added the loop to the experiment then the loop will automatically inform the experiment that it has received data.

Multiple data name/value pairs can be added to any given entry of the data file and is considered part of the same entry until the `nextEntry()` call is made.

e.g.:

```
# add some data for this trial
exp.addData('resp.rt', 0.8)
exp.addData('resp.key', 'k')
# end of trial - move to next line in data output
exp.nextEntry()
```

addLoop (*loopHandler*)

Add a loop such as a *TrialHandler* or *StairHandler* Data from this loop will be included in the resulting data files.

close ()

property currentLoop

Return the loop which we are currently in, this will either be a handle to a loop, such as a *TrialHandler* or *StairHandler*, or the handle of the *ExperimentHandler* itself if we are not in a loop.

getAllEntries ()

Fetches a copy of all the entries including a final (orphan) entry if that exists. This allows entries to be saved even if nextEntry() is not yet called.

Returns copy (not pointer) to entries

loopEnded (*loopHandler*)

Informs the experiment handler that the loop is finished and not to include its values in further entries of the experiment.

This method is called by the loop itself if it ends its iterations, so is not typically needed by the user.

nextEntry ()

Calling nextEntry indicates to the ExperimentHandler that the current trial has ended and so further addData() calls correspond to the next trial.

saveAsPickle (*fileName, fileCollisionMethod='rename'*)

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters fileCollisionMethod: Collision method passed to handleFileCollision ()

saveAsWideText (*fileName, delim='auto', matrixOnly=False, appendFile=None, encoding='utf-8-sig', fileCollisionMethod='rename', sortColumns=False*)

Saves a long, wide-format text file, with one line representing the attributes and data for a single trial. Suitable for analysis in R and SPSS.

If *appendFile=True* then the data will be added to the bottom of an existing file. Otherwise, if the file exists already it will be kept and a new file will be created with a slightly different name. If you want to overwrite the old file, pass 'overwrite' to fileCollisionMethod.

If *matrixOnly=True* then the file will not contain a header row, which can be handy if you want to append data to an existing file of the same format.

Parameters

fileName: if extension is not specified, '.csv' will be appended if the delimiter is ';', else '.tsv' will be appended. Can include path info.

delim: allows the user to use a delimiter other than the default tab (";" is popular with file extension ".csv")

matrixOnly: outputs the data with no header row.

appendFile: will add this output to the end of the specified file if it already exists.

encoding: The encoding to use when saving a the file. Defaults to *utf-8-sig*.

fileCollisionMethod: Collision method passed to handleFileCollision ()

sortColumns: will sort columns alphabetically by header name if True

timestampOnFlip (*win, name*)

Add a timestamp (in the future) to the current row

Parameters

- **win** (*psychopy.visual.Window*) – The window object that we'll base the timestamp flip on
- **name** (*str*) – The name of the column in the datafile being written, such as 'myStim.stopped'

9.12.2 TrialHandler

```
class psychopy.data.TrialHandler (trialList, nReps, method='random', dataTypes=None, extraInfo=None, seed=None, originPath=None, name="", autoLog=True)
```

Class to handle trial sequencing and data storage.

Calls to `.next()` will fetch the next trial object given to this handler, according to the method specified (random, sequential, fullRandom). Calls will raise a StopIteration error if trials have finished.

See `demo_trialHandler.py`

The psydat file format is literally just a pickled copy of the TrialHandler object that saved it. You can open it with:

```
from psychopy.tools.filetools import fromFile
dat = fromFile(path)
```

Then you'll find that `dat` has the following attributes that

Parameters

trialList: a simple list (or flat array) of dictionaries specifying conditions. This can be imported from an excel/csv file using `importConditions()`

nReps: number of repeats for all conditions

method: 'random', 'sequential', or 'fullRandom' 'sequential' obviously presents the conditions in the order they appear in the list. 'random' will result in a shuffle of the conditions on each repeat, but all conditions occur once before the second repeat etc. 'fullRandom' fully randomises the trials across repeats as well, which means you could potentially run all trials of one condition before any trial of another.

dataTypes: (optional) list of names for data storage. e.g. ['corr','rt','resp']. If not provided then these will be created as needed during calls to `addData()`

extraInfo: A dictionary This will be stored alongside the data and usually describes the experiment and subject ID, date etc.

seed: an integer If provided then this fixes the random number generator to use the same pattern of trials, by seeding its startpoint

originPath: a string describing the location of the script / experiment file path. The psydat file format will store a copy of the experiment if possible. If `originPath==None` is provided here then the TrialHandler will still store a copy of the script where it was created. If `OriginPath==-1` then nothing will be stored.

Attributes (after creation)

.data - a dictionary (or more strictly, a DataHandler subclass of a dictionary) of numpy arrays, one for each data type stored

.trialList - the original list of dicts, specifying the conditions

.thisIndex - the index of the current trial in the original conditions list

.nTotal - the total number of trials that will be run

.nRemaining - the total number of trials remaining

.thisN - total trials completed so far

.thisRepN - which repeat you are currently on

.thisTrialN - which trial number *within* that repeat

.thisTrial - a dictionary giving the parameters of the current trial

.finished - True/False for have we finished yet

.extraInfo - the dictionary of extra info as given at beginning

.origin - the contents of the script or builder experiment that created the handler

_createOutputArray (*stimOut, dataOut, delim=None, matrixOnly=False*)

Does the leg-work for saveAsText and saveAsExcel. Combines stimOut with ._parseDataOutput()

_createOutputArrayData (*dataOut*)

This just creates the dataOut part of the output matrix. It is called by _createOutputArray() which creates the header line and adds the stimOut columns

_createSequence ()

Pre-generates the sequence of trial presentations (for non-adaptive methods). This is called automatically when the TrialHandler is initialised so doesn't need an explicit call from the user.

The returned sequence has form indices[stimN][repN] Example: sequential with 6 trialtypes (rows), 5 reps (cols), returns:

```
[[0 0 0 0 0]
[1 1 1 1 1]
[2 2 2 2 2]
[3 3 3 3 3]
[4 4 4 4 4]
[5 5 5 5 5]]
```

These 30 trials will be returned by .next() in the order: 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5

To add a new type of sequence (as of v1.65.02): - add the sequence generation code here - adjust "if self.method in [...]:" in both __init__ and .next() - adjust allowedVals in experiment.py -> shows up in DlgLoopProperties Note that users can make any sequence whatsoever outside of PsychoPy, and specify sequential order; any order is possible this way.

_makeIndices (*inputArray*)

Creates an array of tuples the same shape as the input array where each tuple contains the indices to itself in the array.

Useful for shuffling and then using as a reference.

_terminate ()

Remove references to ourself in experiments and terminate the loop

addData (*thisType, value, position=None*)

Add data for the current trial

getCurrentTrial ()

Returns the condition for the current trial, without advancing the trials.

getEarlierTrial (*n=- 1*)

Returns the condition information from n trials previously. Useful for comparisons in n-back tasks. Returns 'None' if trying to access a trial prior to the first.

getExp ()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getFutureTrial (*n=1*)

Returns the condition for n trials into the future, without advancing the trials. A negative n returns a previous (past) trial. Returns 'None' if attempting to go beyond the last trial.

getOriginPathAndFile (*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (file from a standard python script).

next ()

Advances to next trial and returns it. Updates attributes; thisTrial, thisTrialN and thisIndex. If the trials have ended this method will raise a StopIteration error. This can be handled with code such as:

```
trials = data.TrialHandler(.....)
for eachTrial in trials: # automatically stops when done
    # do stuff
```

or:

```
trials = data.TrialHandler(.....)
while True: # ie forever
    try:
        thisTrial = trials.next()
    except StopIteration: # we got a StopIteration error
        break #break out of the forever loop
    # do stuff here for the trial
```

printAsText (*stimOut=None, dataOut='all_mean', 'all_std', 'all_raw', delim='\t', matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

saveAsExcel (*fileName, sheetName='rawData', stimOut=None, dataOut='n', 'all_mean', 'all_std', 'all_raw', matrixOnly=False, appendFile=True, fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see *TrialHandler.saveAsText()*) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

Parameters

fileName: **string** the name of the file to create or append. Can include relative or absolute path

sheetName: **string** the name of the worksheet within the file

stimOut: **list of strings** the attributes of the trial characteristics to be output. To use this you need to have provided a list of dictionaries specifying to trialList parameter of the TrialHandler and give here the names of strings specifying entries in that dictionary

dataOut: **list of strings** specifying the dataType and the analysis to be performed, in the form *dataType_analysis*. The data can be any of the types that you added using trialHandler.data.add() and the analysis can be either 'raw' or most things in the numpy library,

including 'mean', 'std', 'median', 'max', 'min'. e.g. `rt_max` will give a column of max reaction times across the trials assuming that `rt` values have been stored. The default values will output the raw, mean and std of all datatypes found.

appendFile: True or False If False any existing file with this name will be kept and a new file will be created with a slightly different name. If you want to overwrite the old file, pass 'overwrite' to `fileCollisionMethod`. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string Collision method (`rename`, `overwrite`, `fail`) passed to `handleFileCollision()` This is ignored if `append` is True.

saveAsJson (`fileName=None, encoding='utf-8', fileCollisionMethod='rename'`)
Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of 'rename', 'overwrite', or 'fail'.

Notes

Currently, a copy of the object is created, and the copy's `.origin` attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle (`fileName, fileCollisionMethod='rename'`)
Basically just saves a copy of the handler (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters `fileCollisionMethod`: Collision method passed to `handleFileCollision()`

saveAsText (`fileName, stimOut=None, dataOut='n', 'all_mean', 'all_std', 'all_raw', delim=None, matrixOnly=False, appendFile=True, summarised=True, fileCollisionMethod='rename', encoding='utf-8-sig'`)

Write a text file with the data and various chosen stimulus attributes

Parameters

fileName: will have `.tsv` appended and can include path info.

stimOut: the stimulus attributes to be output. To use this you need to use a list of dictionaries and give here the names of dictionary keys that you want as strings

dataOut: a list of strings specifying the `dataType` and the analysis to be performed, in the form `dataType_analysis`. The data can be any of the types that you added using `trialHandler.data.add()` and the analysis can be either 'raw' or most things in the numpy library, including; 'mean', 'std', 'median', 'max', 'min'... The default values will output the raw, mean and std of all datatypes found

delim: allows the user to use a delimiter other than tab ("," is popular with file extension ".csv")

matrixOnly: outputs the data with no header row or extraInfo attached

appendFile: will add this output to the end of the specified file if it already exists

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to *utf-8-sig*.

saveAsWideText (*fileName*, *delim=None*, *matrixOnly=False*, *appendFile=True*, *encoding='utf-8-sig'*, *fileCollisionMethod='rename'*)

Write a text file with the session, stimulus, and data values from each trial in chronological order. Also, return a pandas DataFrame containing same information as the file.

That is, unlike ‘saveAsText’ and ‘saveAsExcel’:

- each row comprises information from only a single trial.
- no summarizing is done (such as collapsing to produce mean and standard deviation values across trials).

This ‘wide’ format, as expected by R for creating dataframes, and various other analysis programs, means that some information must be repeated on every row.

In particular, if the trialHandler’s ‘extraInfo’ exists, then each entry in there occurs in every row. In builder, this will include any entries in the ‘Experiment info’ field of the ‘Experiment settings’ dialog. In Coder, this information can be set using something like:

```
myTrialHandler.extraInfo = {'SubjID': 'Joan Smith',
                             'Group': 'Control'}
```

Parameters

fileName: if extension is not specified, ‘.csv’ will be appended if the delimiter is ‘;’, else ‘.tsv’ will be appended. Can include path info.

delim: allows the user to use a delimiter other than the default tab (“,”) is popular with file extension “.csv”)

matrixOnly: outputs the data with no header row.

appendFile: will add this output to the end of the specified file if it already exists.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to *utf-8-sig*.

setExp (*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

9.12.3 TrialHandler2

```
class psychopy.data.TrialHandler2(trialList, nReps, method='random', dataTypes=None, extraInfo=None, seed=None, originPath=None, name='', autoLog=True)
```

Class to handle trial sequencing and data storage.

Calls to `.next()` will fetch the next trial object given to this handler, according to the method specified (random, sequential, fullRandom). Calls will raise a `StopIteration` error if trials have finished.

See `demo_trialHandler.py`

The psydat file format is literally just a pickled copy of the `TrialHandler` object that saved it. You can open it with:

```
from psychopy.tools.filetools import fromFile
dat = fromFile(path)
```

Then you'll find that `dat` has the following attributes that

Parameters

trialList: filename or a simple list (or flat array) of dictionaries specifying conditions

nReps: number of repeats for all conditions

method: *'random'*, *'sequential'*, or *'fullRandom'* *'sequential'* obviously presents the conditions in the order they appear in the list. *'random'* will result in a shuffle of the conditions on each repeat, but all conditions occur once before the second repeat etc. *'fullRandom'* fully randomises the trials across repeats as well, which means you could potentially run all trials of one condition before any trial of another.

dataTypes: (optional) list of names for data storage. e.g. [*'corr'*,*'rt'*,*'resp'*]. If not provided then these will be created as needed during calls to `addData()`

extraInfo: A dictionary This will be stored alongside the data and usually describes the experiment and subject ID, date etc.

seed: an integer If provided then this fixes the random number generator to use the same pattern of trials, by seeding its startpoint.

originPath: a string describing the location of the script / experiment file path. The psydat file format will store a copy of the experiment if possible. If `originPath=None` is provided here then the `TrialHandler` will still store a copy of the script where it was created. If `OriginPath=-1` then nothing will be stored.

Attributes (after creation)

.data - a dictionary of numpy arrays, one for each data type stored

.trialList - the original list of dicts, specifying the conditions

.thisIndex - the index of the current trial in the original conditions list

.nTotal - the total number of trials that will be run

.nRemaining - the total number of trials remaining

.thisN - total trials completed so far

.thisRepN - which repeat you are currently on

.thisTrialN - which trial number *within* that repeat

.thisTrial - a dictionary giving the parameters of the current trial

.finished - True/False for have we finished yet

.extraInfo - the dictionary of extra info as given at beginning

.origin - the contents of the script or builder experiment that created the handler

_terminate ()

Remove references to ourself in experiments and terminate the loop

addData (thisType, value)

Add a piece of data to the current trial

property data

Returns a pandas DataFrame of the trial data so far Read only attribute - you can't directly modify TrialHandler.data

Note that data are stored internally as a list of dictionaries, one per trial. These are converted to a DataFrame on access.

getEarlierTrial (n=- 1)

Returns the condition information from n trials previously. Useful for comparisons in n-back tasks. Returns 'None' if trying to access a trial prior to the first.

getExp ()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getFutureTrial (n=1)

Returns the condition for n trials into the future, without advancing the trials. Returns 'None' if attempting to go beyond the last trial.

getOriginPathAndFile (originPath=None)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

next ()

Advances to next trial and returns it. Updates attributes; thisTrial, thisTrialN and thisIndex If the trials have ended this method will raise a StopIteration error. This can be handled with code such as:

```
trials = data.TrialHandler(.....)
for eachTrial in trials: # automatically stops when done
    # do stuff
```

or:

```
trials = data.TrialHandler(.....)
while True: # ie forever
    try:
        thisTrial = trials.next()
    except StopIteration: # we got a StopIteration error
        break # break out of the forever loop
    # do stuff here for the trial
```

printAsText (stimOut=None, dataOut='all_mean', 'all_std', 'all_raw', delim='\t', matrixOnly=False)

Exactly like saveAsText() except that the output goes to the screen instead of a file

saveAsExcel (fileName, sheetName='rawData', stimOut=None, dataOut='n', 'all_mean', 'all_std', 'all_raw', matrixOnly=False, appendFile=True, fileCollisionMethod='rename')

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet

packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see `TrialHandler.saveAsText()`) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension `.xlsx` will be added if not given already.

Parameters

fileName: **string** the name of the file to create or append. Can include relative or absolute path

sheetName: **string** the name of the worksheet within the file

stimOut: **list of strings** the attributes of the trial characteristics to be output. To use this you need to have provided a list of dictionaries specifying to `trialList` parameter of the `TrialHandler` and give here the names of strings specifying entries in that dictionary

dataOut: **list of strings** specifying the `dataType` and the analysis to be performed, in the form `dataType_analysis`. The data can be any of the types that you added using `trialHandler.data.add()` and the analysis can be either 'raw' or most things in the numpy library, including 'mean', 'std', 'median', 'max', 'min'. e.g. `rt_max` will give a column of max reaction times across the trials assuming that `rt` values have been stored. The default values will output the raw, mean and std of all datatypes found.

appendFile: **True or False** If False any existing file with this name will be kept and a new file will be created with a slightly different name. If you want to overwrite the old file, pass 'overwrite' to `fileCollisionMethod`. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: **string** Collision method (`rename`, `overwrite`, `fail`) passed to `handleFileCollision()` This is ignored if `append` is True.

saveAsJson (`fileName=None`, `encoding='utf-8'`, `fileCollisionMethod='rename'`)

Serialize the object to the JSON format.

Parameters

- **fileName** (`string`, or `None`) – the name of the file to create or append. Can include a relative or absolute path. If `None`, will not write to a file, but return an in-memory JSON object.
- **encoding** (`string`, optional) – The encoding to use when writing the file.
- **fileCollisionMethod** (`string`) – Collision method passed to `handleFileCollision()`. Can be either of 'rename', 'overwrite', or 'fail'.

Notes

Currently, a copy of the object is created, and the copy's `.origin` attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

The RNG `self._rng` cannot be serialized as-is, so we store its state in `self._rng_state` so we can restore it when loading.

saveAsPickle (*fileName*, *fileCollisionMethod*='rename')

Basically just saves a copy of the handler (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters `fileCollisionMethod`: Collision method passed to `handleFileCollision()`

saveAsText (*fileName*, *stimOut*=None, *dataOut*='n', 'all_mean', 'all_std', 'all_raw', *delim*=None, *matrixOnly*=False, *appendFile*=True, *summarised*=True, *fileCollisionMethod*='rename', *encoding*='utf-8-sig')

Write a text file with the data and various chosen stimulus attributes

Parameters

fileName: will have `.tsv` appended and can include path info.

stimOut: the stimulus attributes to be output. To use this you need to use a list of dictionaries and give here the names of dictionary keys that you want as strings

dataOut: a list of strings specifying the `dataType` and the analysis to be performed, in the form `dataType_analysis`. The data can be any of the types that you added using `trialHandler.data.add()` and the analysis can be either 'raw' or most things in the numpy library, including: 'mean', 'std', 'median', 'max', 'min'... The default values will output the raw, mean and std of all datatypes found

delim: allows the user to use a delimiter other than tab ("`,`" is popular with file extension `".csv"`)

matrixOnly: outputs the data with no header row or extraInfo attached

appendFile: will add this output to the end of the specified file if it already exists

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to `utf-8-sig`.

saveAsWideText (*fileName*, *delim*=None, *matrixOnly*=False, *appendFile*=True, *encoding*='utf-8-sig', *fileCollisionMethod*='rename')

Write a text file with the session, stimulus, and data values from each trial in chronological order. Also, return a pandas DataFrame containing same information as the file.

That is, unlike 'saveAsText' and 'saveAsExcel':

- each row comprises information from only a single trial.
- no summarising is done (such as collapsing to produce mean and standard deviation values across trials).

This 'wide' format, as expected by R for creating dataframes, and various other analysis programs, means that some information must be repeated on every row.

In particular, if the `trialHandler`'s `'extraInfo'` exists, then each entry in there occurs in every row. In builder, this will include any entries in the 'Experiment info' field of the 'Experiment settings' dialog. In Coder, this information can be set using something like:

```
myTrialHandler.extraInfo = {'SubjID': 'Joan Smith',
                             'Group': 'Control'}
```

Parameters

fileName: if extension is not specified, ‘.csv’ will be appended if the delimiter is ‘,’, else ‘.tsv’ will be appended. Can include path info.

delim: allows the user to use a delimiter other than the default tab (“,” is popular with file extension “.csv”)

matrixOnly: outputs the data with no header row.

appendFile: will add this output to the end of the specified file if it already exists.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to *utf-8-sig*.

setExp (*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

9.12.4 TrialHandlerExt

class `psychopy.data.TrialHandlerExt` (*trialList, nReps, method='random', dataTypes=None, extraInfo=None, seed=None, originPath=None, name="", autoLog=True*)

A class for handling trial sequences in a *non-counterbalanced design* (i.e. *oddball paradigms*). Its functions are a superset of the class `TrialHandler`, and as such, can also be used for normal trial handling.

`TrialHandlerExt` has the same function names for data storage facilities.

To use non-counterbalanced designs, all `TrialType` dict entries in the trial list must have a key called “weight”. For example, if you want trial types A, B, C, and D to have 10, 5, 3, and 2 repetitions per block, then the trialList can look like:

```
[[Name:'A', ..., weight:10], {Name:'B', ..., weight:5}, {Name:'C', ..., weight:3}, {Name:'D', ..., weight:2}]
```

For experimenters using an excel or csv file for trial list, a column called weight is appropriate for this purpose.

Calls to `.next()` will fetch the next trial object given to this handler, according to the method specified (`random`, `sequential`, `fullRandom`). Calls will raise a `StopIteration` error when all trials are exhausted.

Authored by Suddha Sourav at BPN, Uni Hamburg - heavily borrowing from the TrialHandler class

Parameters

trialList: a simple list (or flat array) of dictionaries specifying conditions. This can be imported from an excel / csv file using `importConditions()` For non-counterbalanced designs, each dict entry in trialList must have a key called weight!

nReps: number of repeats for all conditions. When using a non-counterbalanced design, nReps is analogous to the number of blocks.

method: *'random'*, *'sequential'*, or *'fullRandom'* When the weights are not specified: *'sequential'* presents the conditions in the order they appear in the list. *'random'* will result in a shuffle of the conditions on each repeat, but all conditions occur once before the second repeat etc. *'fullRandom'* fully randomises the trials across repeats as well, which means you could potentially run all trials of one condition before any trial of another.

In the presence of weights: *'sequential'* presents each trial type the number of times specified by its weight, before moving on to the next type. *'random'* randomizes the presentation order within block. *'fullRandom'* shuffles trial order across weights an nRep, that is, a full shuffling.

dataTypes: (optional) list of names for data storage. e.g. [*'corr'*,*'rt'*,*'resp'*]. If not provided then these will be created as needed during calls to *addData()*

extraInfo: A dictionary This will be stored alongside the data and usually describes the experiment and subject ID, date etc.

seed: an integer If provided then this fixes the random number generator to use the same pattern of trials, by seeding its startpoint

originPath: a string describing the location of the script / experiment file path. The psydat file format will store a copy of the experiment if possible. If *originPath==None* is provided here then the TrialHandler will still store a copy of the script where it was created. If *OriginPath==-1* then nothing will be stored.

Attributes (after creation)

.data - a dictionary of numpy arrays, one for each data type stored

.trialList - the original list of dicts, specifying the conditions

.thisIndex - the index of the current trial in the original conditions list

.nTotal - the total number of trials that will be run

.nRemaining - the total number of trials remaining

.thisN - total trials completed so far

.thisRepN - which repeat you are currently on

.thisTrialN - which trial number *within* that repeat

.thisTrial - a dictionary giving the parameters of the current trial

.finished - True/False for have we finished yet

.extraInfo - the dictionary of extra info as given at beginning

.origin - the contents of the script or builder experiment that created the handler

.trialWeights - None if all weights are not specified. If all weights are specified, then a list containing the weights of the trial types.

_createOutputArray (*stimOut, dataOut, delim=None, matrixOnly=False*)

Does the leg-work for *saveAsText* and *saveAsExcel*. Combines *stimOut* with *._parseDataOutput()*

_createOutputArrayData (*dataOut*)

This just creates the *dataOut* part of the output matrix. It is called by *_createOutputArray()* which creates the header line and adds the *stimOut* columns

_createSequence ()

Pre-generates the sequence of trial presentations (for non-adaptive methods). This is called automatically when the TrialHandler is initialised so doesn't need an explicit call from the user.

The returned sequence has form indices[stimN][repN] Example: sequential with 6 trialtypes (rows), 5 reps (cols), returns:

```
[[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]
 [5 5 5 5 5]]
```

These 30 trials will be returned by .next() in the order: 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5

Example: random, with 3 trialtypes, where the weights of conditions 0,1, and 2 are 3,2, and 1 respectively, and a rep value of 5, might return:

```
[[0 1 2 0 1]
 [1 0 1 1 1]
 [0 2 0 0 0]
 [0 0 0 1 0]
 [2 0 1 0 2]
 [1 1 0 2 0]]
```

These 30 trials will be returned by .next() in the order: 0, 1, 0, 0, 2, 1, 1, 0, 2, 0, 0, 1, 0, 2, 0
stopIteration

To add a new type of sequence (as of v1.65.02): - add the sequence generation code here - adjust “if self.method in [...]:” in both `__init__` and `.next()` - adjust `allowedVals` in `experiment.py` -> shows up in `DlgLoopProperties` Note that users can make any sequence whatsoever outside of PsychoPy, and specify sequential order; any order is possible this way.

`_makeIndices (inputArray)`

Creates an array of tuples the same shape as the input array where each tuple contains the indices to itself in the array.

Useful for shuffling and then using as a reference.

`_terminate ()`

Remove references to ourself in experiments and terminate the loop

`addData (thisType, value, position=None)`

Add data for the current trial

`getCurrentTrial ()`

Returns the condition for the current trial, without advancing the trials.

`getCurrentTrialPosInDataHandler ()`

`getEarlierTrial (n=- 1)`

Returns the condition information from n trials previously. Useful for comparisons in n-back tasks. Returns ‘None’ if trying to access a trial prior to the first.

`getExp ()`

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

`getFutureTrial (n=1)`

Returns the condition for n trials into the future, without advancing the trials. A negative n returns a previous (past) trial. Returns ‘None’ if attempting to go beyond the last trial.

`getNextTrialPosInDataHandler ()`

getOriginPathAndFile (*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If *originPath* is provided (e.g. from Builder) then this is used otherwise the calling script is the *originPath* (fine from a standard python script).

next ()

Advances to next trial and returns it. Updates attributes; *thisTrial*, *thisTrialN* and *thisIndex* If the trials have ended this method will raise a *StopIteration* error. This can be handled with code such as:

```
trials = data.TrialHandler(...)
for eachTrial in trials: # automatically stops when done
    # do stuff
```

or:

```
trials = data.TrialHandler(...)
while True: # ie forever
    try:
        thisTrial = trials.next()
    except StopIteration: # we got a StopIteration error
        break # break out of the forever loop
    # do stuff here for the trial
```

printAsText (*stimOut=None, dataOut='all_mean', 'all_std', 'all_raw', delim='\t', matrixOnly=False*)

Exactly like *saveAsText*() except that the output goes to the screen instead of a file

saveAsExcel (*fileName, sheetName='rawData', stimOut=None, dataOut='n', 'all_mean', 'all_std', 'all_raw', matrixOnly=False, appendFile=True, fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see *TrialHandler.saveAsText()*) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

Parameters

fileName: **string** the name of the file to create or append. Can include relative or absolute path

sheetName: **string** the name of the worksheet within the file

stimOut: **list of strings** the attributes of the trial characteristics to be output. To use this you need to have provided a list of dictionaries specifying to *trialList* parameter of the *TrialHandler* and give here the names of strings specifying entries in that dictionary

dataOut: **list of strings** specifying the *dataType* and the analysis to be performed, in the form *dataType_analysis*. The data can be any of the types that you added using *trialHandler.data.add()* and the analysis can be either 'raw' or most things in the numpy library, including 'mean', 'std', 'median', 'max', 'min'. e.g. *rt_max* will give a column of max reaction times across the trials assuming that *rt* values have been stored. The default values will output the raw, mean and std of all datatypes found.

appendFile: **True or False** If False any existing file with this name will be kept and a new file will be created with a slightly different name. If you want to overwrite the old file,

pass 'overwrite' to `fileCollisionMethod`. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: `string` Collision method (`rename`, `'overwrite'`, `fail`) passed to `handleFileCollision()` This is ignored if `append` is True.

saveAsJson (`fileName=None`, `encoding='utf-8'`, `fileCollisionMethod='rename'`)
Serialize the object to the JSON format.

Parameters

- **fileName** (`string`, or `None`) – the name of the file to create or append. Can include a relative or absolute path. If `None`, will not write to a file, but return an in-memory JSON object.
- **encoding** (`string`, optional) – The encoding to use when writing the file.
- **fileCollisionMethod** (`string`) – Collision method passed to `handleFileCollision()`. Can be either of `'rename'`, `'overwrite'`, or `'fail'`.

Notes

Currently, a copy of the object is created, and the copy's `.origin` attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle (`fileName`, `fileCollisionMethod='rename'`)
Basically just saves a copy of the handler (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters `fileCollisionMethod`: Collision method passed to `handleFileCollision()`

saveAsText (`fileName`, `stimOut=None`, `dataOut='n'`, `'all_mean'`, `'all_std'`, `'all_raw'`, `delim=None`, `matrixOnly=False`, `appendFile=True`, `summarised=True`, `fileCollisionMethod='rename'`, `encoding='utf-8-sig'`)

Write a text file with the data and various chosen stimulus attributes

Parameters

fileName: will have `.tsv` appended and can include path info.

stimOut: the stimulus attributes to be output. To use this you need to use a list of dictionaries and give here the names of dictionary keys that you want as strings

dataOut: a list of strings specifying the `dataType` and the analysis to be performed, in the form `dataType_analysis`. The data can be any of the types that you added using `trialHandler.data.add()` and the analysis can be either `'raw'` or most things in the numpy library, including: `'mean'`, `'std'`, `'median'`, `'max'`, `'min'`... The default values will output the raw, mean and std of all datatypes found

delim: allows the user to use a delimiter other than tab ("`,`" is popular with file extension `".csv"`)

matrixOnly: outputs the data with no header row or `extraInfo` attached

appendFile: will add this output to the end of the specified file if it already exists

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to `utf-8-sig`.

saveAsWideText (*fileName*, *delim*='\t', *matrixOnly*=False, *appendFile*=True, *encoding*='utf-8-sig', *fileCollisionMethod*='rename')

Write a text file with the session, stimulus, and data values from each trial in chronological order.

That is, unlike ‘saveAsText’ and ‘saveAsExcel’:

- each row comprises information from only a single trial.
- no summarizing is done (such as collapsing to produce mean and standard deviation values across trials).

This ‘wide’ format, as expected by R for creating dataframes, and various other analysis programs, means that some information must be repeated on every row.

In particular, if the trialHandler’s ‘extraInfo’ exists, then each entry in there occurs in every row. In builder, this will include any entries in the ‘Experiment info’ field of the ‘Experiment settings’ dialog. In Coder, this information can be set using something like:

```
myTrialHandler.extraInfo = {'SubjID': 'Joan Smith',
                             'Group': 'Control'}
```

Parameters

fileName: if extension is not specified, ‘.csv’ will be appended if the delimiter is ‘;’, else ‘.txt’ will be appended. Can include path info.

delim: allows the user to use a delimiter other than the default tab (“\t” is popular with file extension “.csv”)

matrixOnly: outputs the data with no header row.

appendFile: will add this output to the end of the specified file if it already exists.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to *utf-8-sig*.

setExp (*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

9.12.5 StairHandler

class psychopy.data.**StairHandler** (*startVal*, *nReversals*=None, *stepSizes*=4, *nTrials*=0, *nUp*=1, *nDown*=3, *applyInitialRule*=True, *extraInfo*=None, *method*='2AFC', *stepType*='db', *minVal*=None, *maxVal*=None, *originPath*=None, *name*='', *autoLog*=True, ***kwargs*)

Class to handle smoothly the selection of the next trial and report current values etc. Calls to next() will fetch the next object given to this handler, according to the method specified.

See Demos >> ExperimentalControl >> JND_staircase_exp.py

The staircase will terminate when *nTrials* AND *nReversals* have been exceeded. If *stepSizes* was an array and has been exceeded before *nTrials* is exceeded then the staircase will continue to reverse.

nUp and *nDown* are always considered as 1 until the first reversal is reached. The values entered as arguments are then used.

Parameters

startVal: The initial value for the staircase.

nReversals: The minimum number of reversals permitted. If *stepSizes* is a list, but the minimum number of reversals to perform, *nReversals*, is less than the length of this list, PsychoPy will automatically increase the minimum number of reversals and emit a warning.

stepSizes: The size of steps as a single value or a list (or array). For a single value the step size is fixed. For an array or list the step size will progress to the next entry at each reversal.

nTrials: The minimum number of trials to be conducted. If the staircase has not reached the required number of reversals then it will continue.

nUp: The number of 'incorrect' (or 0) responses before the staircase level increases.

nDown: The number of 'correct' (or 1) responses before the staircase level decreases.

applyInitialRule [bool] Whether to apply a 1-up/1-down rule until the first reversal point (if *True*), before switching to the specified up/down rule.

extraInfo: A dictionary (typically) that will be stored along with collected data using *saveAsPickle()* or *saveAsText()* methods.

method: Not used and may be deprecated in future releases.

stepType: 'db', 'lin', 'log' The type of steps that should be taken each time. 'lin' will simply add or subtract that amount each step, 'db' and 'log' will step by a certain number of decibels or log units (note that this will prevent your value ever reaching zero or less)

minVal: *None*, or a number The smallest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

maxVal: *None*, or a number The largest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

Additional keyword arguments will be ignored.

Notes

The additional keyword arguments ***kwargs* might for example be passed by the *MultiStairHandler*, which expects a *label* keyword for each staircase. These parameters are to be ignored by the *StairHandler*.

`__intensityDec()`

decrement the current intensity and reset counter

`__intensityInc()`

increment the current intensity and reset counter

`__terminate()`

Remove references to ourself in experiments and terminate the loop

`addData(result, intensity=None)`

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

- `.addResponse(result, intensity)`
- `.addOtherData('dataName', value')`

`addOtherData(dataName, value)`

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

addResponse (*result, intensity=None*)

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial.

This is essential to advance the staircase to a new intensity level!

Supplying an *intensity* value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

calculateNextIntensity ()

Based on current intensity, counter of correct responses, and current direction.

getExp ()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getOriginPathAndFile (*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If *originPath* is provided (e.g. from Builder) then this is used otherwise the calling script is the *originPath* (fine from a standard python script).

property intensity

The intensity (level) of the current staircase

next ()

Advances to next trial and returns it. Updates attributes; *thisTrial*, *thisTrialN* and *thisIndex*.

If the trials have ended, calling this method will raise a *StopIteration* error. This can be handled with code such as:

```
staircase = data.StairHandler(.....)
for eachTrial in staircase: # automatically stops when done
    # do stuff
```

or:

```
staircase = data.StairHandler(.....)
while True: # ie forever
    try:
        thisTrial = staircase.next()
    except StopIteration: # we got a StopIteration error
        break # break out of the forever loop
    # do stuff here for the trial
```

printAsText (*stimOut=None, dataOut='all_mean', 'all_std', 'all_raw', delim='\t', matrixOnly=False*)

Exactly like *saveAsText*() except that the output goes to the screen instead of a file

saveAsExcel (*fileName, sheetName='data', matrixOnly=False, appendFile=True, fileCollision-Method='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see *TrialHandler.saveAsText()*) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: **string** the name of the file to create or append. Can include relative or absolute path.

sheetName: **string** the name of the worksheet within the file

matrixOnly: **True or False** If set to True then only the data itself will be output (no additional info)

appendFile: **True or False** If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: **string** Collision method passed to `handleFileCollision()`
This is ignored if `appendFile` is True.

saveAsJson (*fileName=None, encoding='utf-8-sig', fileCollisionMethod='rename'*)
Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's `.origin` attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle (*fileName, fileCollisionMethod='rename'*)
Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters `fileCollisionMethod`: Collision method passed to `handleFileCollision()`

saveAsText (*fileName, delim=None, matrixOnly=False, fileCollisionMethod='rename', encoding='utf-8-sig'*)
Write a text file with the data

Parameters

fileName: **a string** The name of the file, including path if needed. The extension `.tsv` will be added if not included.

delim: **a string** the delimiter to be used (e.g. `'\t'` for tab-delimited, `'\n'` for csv files)

matrixOnly: **True/False** If True, prevents the output of the `extraInfo` provided at initialisation.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to `utf-8-sig`.

setExp (*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

9.12.6 PsiHandler

class psychopy.data.PsiHandler (*nTrials, intensRange, alphaRange, betaRange, intensPrecision, alphaPrecision, betaPrecision, delta, stepType='lin', expectedMin=0.5, prior=None, fromFile=False, extraInfo=None, name=""*)

Handler to implement the “Psi” adaptive psychophysical method (Kontsevich & Tyler, 1999).

This implementation assumes the form of the psychometric function to be a cumulative Gaussian. Psi estimates the two free parameters of the psychometric function, the location (alpha) and slope (beta), using Bayes’ rule and grid approximation of the posterior distribution. It chooses stimuli to present by minimizing the entropy of this grid. Because this grid is represented internally as a 4-D array, one must choose the intensity, alpha, and beta ranges carefully so as to avoid a Memory Error. Maximum likelihood is used to estimate Lambda, the most likely location/slope pair. Because Psi estimates the entire psychometric function, any threshold defined on the function may be estimated once Lambda is determined.

It is advised that Lambda estimates are examined after completion of the Psi procedure. If the estimated alpha or beta values equal your specified search bounds, then the search range most likely did not contain the true value. In this situation the procedure should be repeated with appropriately adjusted bounds.

Because Psi is a Bayesian method, it can be initialized with a prior from existing research. A function to save the posterior over Lambda as a Numpy binary file is included.

Kontsevich & Tyler (1999) specify their psychometric function in terms of d' . PsiHandler avoids this and treats all parameters with respect to stimulus intensity. Specifically, the forms of the psychometric function assumed for Yes/No and Two Alternative Forced Choice (2AFC) are, respectively:

$$_normCdf = \text{norm.cdf}(x, \text{mean}=\alpha, \text{sd}=\beta) \quad Y(x) = .5 * \text{delta} + (1 - \text{delta}) * _normCdf$$

$$Y(x) = .5 * \text{delta} + (1 - \text{delta}) * (.5 + .5 * _normCdf)$$

Initializes the handler and creates an internal Psi Object for grid approximation.

Parameters

- nTrials (int)** The number of trials to run.
- intensRange (list)** Two element list containing the (inclusive) endpoints of the stimuli intensity range.
- alphaRange (list)** Two element list containing the (inclusive) endpoints of the alpha (location parameter) range.
- betaRange (list)** Two element list containing the (inclusive) endpoints of the beta (slope parameter) range.
- intensPrecision (float or int)** If `stepType == 'lin'`, this specifies the step size of the stimuli intensity range. If `stepType == 'log'`, this specifies the number of steps in the stimuli intensity range.
- alphaPrecision (float)** The step size of the alpha (location parameter) range.
- betaPrecision (float)** The step size of the beta (slope parameter) range.

delta (float) The guess rate.

stepType (str) The type of steps to be used when constructing the stimuli intensity range. If 'lin' then evenly spaced steps are used. If 'log' then logarithmically spaced steps are used. Defaults to 'lin'.

expectedMin (float) The expected lower asymptote of the psychometric function (PMF).

For a Yes/No task, the PMF usually extends across the interval [0, 1]; here, *expectedMin* should be set to 0.

For a 2-AFC task, the PMF spreads out across [0.5, 1.0]. Therefore, *expectedMin* should be set to 0.5 in this case, and the 2-AFC psychometric function described above going to be is used.

Currently, only Yes/No and 2-AFC designs are supported.

Defaults to 0.5, or a 2-AFC task.

prior (numpy ndarray or str) Optional prior distribution with which to initialize the Psi Object. This can either be a numpy ndarray object or the path to a numpy binary file (.npy) containing the ndarray.

fromFile (str) Flag specifying whether prior is a file pathname or not.

extraInfo (dict) Optional dictionary object used in PsychoPy's built-in logging system.

name (str) Optional name for the PsiHandler used in PsychoPy's built-in logging system.

Raises

NotImplementedError If the supplied *minVal* parameter implies an experimental design other than Yes/No or 2-AFC.

`__checkFinished()`

checks if we are finished. Updates attribute: *finished*

`__intensityDec()`

decrement the current intensity and reset counter

`__intensityInc()`

increment the current intensity and reset counter

`__terminate()`

Remove references to ourselves in experiments and terminate the loop

`addData(result, intensity=None)`

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

- `.addResponse(result, intensity)`
- `.addOtherData('dataName', value')`

`addOtherData(dataName, value)`

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

`addResponse(result, intensity=None)`

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial. Supplying an *intensity* value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

`calculateNextIntensity()`

Based on current intensity, counter of correct responses, and current direction.

estimateLambda ()

Returns a tuple of (location, slope)

estimateThreshold (*thresh, lamb=None*)

Returns an intensity estimate for the provided probability.

The optional argument 'lamb' allows thresholds to be estimated without having to recompute the maximum likelihood lambda.

getExp ()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getOriginPathAndFile (*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If originPath is provided (e.g. from Builder) then this is used otherwise the calling script is the originPath (fine from a standard python script).

property intensity

The intensity (level) of the current staircase

next ()

Advances to next trial and returns it.

printAsText (*stimOut=None, dataOut='all_mean', 'all_std', 'all_raw', delim='\t', matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

saveAsExcel (*fileName, sheetName='data', matrixOnly=False, appendFile=True, fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see *TrialHandler.saveAsText()*) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: string the name of the file to create or append. Can include relative or absolute path.

sheetName: string the name of the worksheet within the file

matrixOnly: True or False If set to True then only the data itself will be output (no additional info)

appendFile: True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string Collision method passed to *handleFileCollision()*
This is ignored if *appendFile* is True.

saveAsJson (*fileName=None, encoding='utf-8-sig', fileCollisionMethod='rename'*)

Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's `.origin` attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle (*fileName, fileCollisionMethod='rename'*)

Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters `fileCollisionMethod`: Collision method passed to `handleFileCollision()`

saveAsText (*fileName, delim=None, matrixOnly=False, fileCollisionMethod='rename', encoding='utf-8-sig'*)

Write a text file with the data

Parameters

fileName: a string The name of the file, including path if needed. The extension `.tsv` will be added if not included.

delim: a string the delimiter to be used (e.g. `' '` for tab-delimited, `' '` for csv files)

matrixOnly: True/False If True, prevents the output of the `extraInfo` provided at initialisation.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to `utf-8-sig`.

savePosterior (*fileName, fileCollisionMethod='rename'*)

Saves the posterior array over `probLambda` as a pickle file with the specified name.

Parameters `fileCollisionMethod` (*string*) – Collision method passed to `handleFileCollision()`

setExp (*exp*)

Sets the `ExperimentHandler` that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the `weakref` module.

9.12.7 QuestHandler

class psychopy.data.QuestHandler (*startVal, startValSd, pThreshold=0.82, nTrials=None, stopInterval=None, method='quantile', beta=3.5, delta=0.01, gamma=0.5, grain=0.01, range=None, extraInfo=None, minVal=None, maxVal=None, staircase=None, originPath=None, name='', autoLog=True, **kwargs*)

Class that implements the Quest algorithm for quick measurement of psychophysical thresholds.

Uses Andrew Straw's [QUEST](#), which is a Python port of Denis Pelli's Matlab code.

Measures threshold using a Weibull psychometric function. Currently, it is not possible to use a different psychometric function.

The Weibull psychometric function is given by the formula

$$\Psi(x) = \delta\gamma + (1 - \delta)[1 - (1 - \gamma) \exp(-10^{\beta(x-T+\epsilon)})]$$

Here, x is an intensity or a contrast (in log10 units), and T is estimated threshold.

Quest internally shifts the psychometric function such that intensity at the user-specified threshold performance level `pThreshold` (e.g., 50% in a yes-no or 75% in a 2-AFC task) is equal to 0. The parameter ϵ is responsible for this shift, and is determined automatically based on the specified `pThreshold` value. It is the parameter Watson & Pelli (1983) introduced to perform measurements at the "optimal sweat factor". Assuming your `QuestHandler` instance is called `q`, you can retrieve this value via `q.epsilon`.

Example:

```
# setup display/window
...
# create stimulus
stimulus = visual.RadialStim(win=win, tex='sinXsin', size=1,
                             pos=[0,0], units='deg')
...
# create staircase object
# trying to find out the point where subject's response is 50 / 50
# if wanted to do a 2AFC then the defaults for pThreshold and gamma
# are good. As start value, we'll use 50% contrast, with SD = 20%
staircase = data.QuestHandler(0.5, 0.2,
                              pThreshold=0.63, gamma=0.01,
                              nTrials=20, minVal=0, maxVal=1)
...
while thisContrast in staircase:
    # setup stimulus
    stimulus.setContrast(thisContrast)
    stimulus.draw()
    win.flip()
    core.wait(0.5)
    # get response
    ...
    # inform QUEST of the response, needed to calculate next level
    staircase.addResponse(thisResp)
...
# can now access 1 of 3 suggested threshold levels
staircase.mean()
staircase.mode()
staircase.quantile(0.5) # gets the median
```

Typical values for `pThreshold` are:

- 0.82 which is equivalent to a 3 up 1 down standard staircase
- **0.63 which is equivalent to a 1 up 1 down standard staircase** (and might want $\gamma=0.01$)

The variable(s) `nTrials` and/or `stopSd` must be specified.

beta, *delta*, and *gamma* are the parameters of the Weibull psychometric function.

Parameters

startVal: Prior threshold estimate or your initial guess threshold.

startValSd: Standard deviation of your starting guess threshold. Be generous with the sd as QUEST will have trouble finding the true threshold if it's more than one sd from your initial guess.

pThreshold Your threshold criterion expressed as probability of response==1. An intensity offset is introduced into the psychometric function so that the threshold (i.e., the midpoint of the table) yields `pThreshold`.

nTrials: *None* or a number The maximum number of trials to be conducted.

stopInterval: *None* or a number The minimum 5-95% confidence interval required in the threshold estimate before stopping. If both this and `nTrials` is specified, whichever happens first will determine when Quest will stop.

method: *'quantile'*, *'mean'*, *'mode'* The method used to determine the next threshold to test. If you want to get a specific threshold level at the end of your staircasing, please use the `quantile`, `mean`, and `mode` methods directly.

beta: *3.5* or a number Controls the steepness of the psychometric function.

delta: *0.01* or a number The fraction of trials on which the observer presses blindly.

gamma: *0.5* or a number The fraction of trials that will generate response 1 when intensity=-Inf.

grain: *0.01* or a number The quantization of the internal table.

range: *None*, or a number The intensity difference between the largest and smallest intensity that the internal table can store. This interval will be centered on the initial guess `tGuess`. QUEST assumes that intensities outside of this range have zero prior probability (i.e., they are impossible).

extraInfo: A dictionary (typically) that will be stored along with collected data using `saveAsPickle()` or `saveAsText()` methods.

minVal: *None*, or a number The smallest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

maxVal: *None*, or a number The largest legal value for the staircase, which can be used to prevent it reaching impossible contrast values, for instance.

staircase: *None* or `StairHandler` Can supply a staircase object with intensities and results. Might be useful to give the quest algorithm more information if you have it. You can also call the `importData` function directly.

Additional keyword arguments will be ignored.

Notes

The additional keyword arguments `**kwargs` might for example be passed by the `MultiStairHandler`, which expects a `label` keyword for each staircase. These parameters are to be ignored by the `StairHandler`.

`_checkFinished()`
checks if we are finished Updates attribute: *finished*

`_intensity()`
assigns the next intensity level

`_intensityDec()`
decrement the current intensity and reset counter

`_intensityInc()`
increment the current intensity and reset counter

`_terminate()`
Remove references to ourself in experiments and terminate the loop

`addData(result, intensity=None)`
Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

- `.addResponse(result, intensity)`
- `.addOtherData('dataName', value')`

`addOtherData(dataName, value)`
Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

`addResponse(result, intensity=None)`
Add a 1 or 0 to signify a correct / detected or incorrect / missed trial

Supplying an *intensity* value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

property `beta`

`calculateNextIntensity()`
based on current intensity and counter of correct responses

`confInterval(getDifference=False)`
Return estimate for the 5%–95% confidence interval (CI).

Parameters

`getDifference (bool)` If `True`, return the width of the confidence interval (95% - 5% percentiles). If `False`, return a NumPy array with estimates for the 5% and 95% boundaries.

Returns scalar or array of length 2.

property `delta`

property `epsilon`

property `gamma`

`getExp()`
Return the ExperimentHandler that this handler is attached to, if any. Returns `None` if not attached

`getOriginPathAndFile(originPath=None)`
Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If `originPath` is provided (e.g. from Builder) then this is used otherwise the calling script is the `originPath` (fine from a standard python script).

property `grain`

importData (*intensities, results*)

import some data which wasn't previously given to the quest algorithm

incTrials (*nNewTrials*)

increase maximum number of trials Updates attribute: *nTrials*

property intensity

The intensity (level) of the current staircase

mean ()

mean of Quest posterior pdf

mode ()

mode of Quest posterior pdf

next ()

Advances to next trial and returns it. Updates attributes; *thisTrial, thisTrialN, thisIndex, finished, intensities*

If the trials have ended, calling this method will raise a `StopIteration` error. This can be handled with code such as:

```
staircase = data.QuestHandler(.....)
for eachTrial in staircase: # automatically stops when done
    # do stuff
```

or:

```
staircase = data.QuestHandler(.....)
while True: # i.e. forever
    try:
        thisTrial = staircase.next()
    except StopIteration: # we got a StopIteration error
        break # break out of the forever loop
    # do stuff here for the trial
```

printAsText (*stimOut=None, dataOut='all_mean', 'all_std', 'all_raw', delim='\t', matrixOnly=False*)

Exactly like `saveAsText()` except that the output goes to the screen instead of a file

quantile (*p=None*)

quantile of Quest posterior pdf

property range

saveAsExcel (*fileName, sheetName='data', matrixOnly=False, appendFile=True, fileCollision-Method='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see `TrialHandler.saveAsText()`) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: **string** the name of the file to create or append. Can include relative or absolute path.

sheetName: **string** the name of the worksheet within the file

matrixOnly: **True or False** If set to True then only the data itself will be output (no additional info)

appendFile: **True or False** If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: **string** Collision method passed to `handleFileCollision()`
This is ignored if `appendFile` is True.

saveAsJson (*fileName=None, encoding='utf-8-sig', fileCollisionMethod='rename'*)
Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's `.origin` attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle (*fileName, fileCollisionMethod='rename'*)
Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters `fileCollisionMethod`: Collision method passed to `handleFileCollision()`

saveAsText (*fileName, delim=None, matrixOnly=False, fileCollisionMethod='rename', encoding='utf-8-sig'*)
Write a text file with the data

Parameters

fileName: **a string** The name of the file, including path if needed. The extension `.tsv` will be added if not included.

delim: **a string** the delimiter to be used (e.g. `'\t'` for tab-delimited, `','` for csv files)

matrixOnly: **True/False** If True, prevents the output of the *extraInfo* provided at initialisation.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to *utf-8-sig*.

sd()
standard deviation of Quest posterior pdf

setExp (*exp*)

Sets the ExperimentHandler that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the *weakref* module.

simulate (*tActual*)

returns a simulated user response to the next intensity level presented by Quest, need to supply the actual threshold level

9.12.8 QuestPlusHandler

```
class psychopy.data.QuestPlusHandler (nTrials, intensityVals, thresholdVals, slopeVals, lowerAsymptoteVals, lapseRateVals, responseVals='Yes', 'No', prior=None, startIntensity=None, psychometricFunc='weibull', stimScale='log10', stimSelectionMethod='minEntropy', stimSelectionOptions=None, paramEstimationMethod='mean', extraInfo=None, name="", label="", **kwargs)
```

QUEST+ implementation. Currently only supports parameter estimation of a Weibull-shaped psychometric function.

The parameter estimates can be retrieved via the *.paramEstimate* attribute, which returns a dictionary whose keys correspond to the names of the estimated parameters (i.e., *QuestPlusHandler.paramEstimate['threshold']* will provide the threshold estimate). Retrieval of the marginal posterior distributions works similarly: they can be accessed via the *.posterior* dictionary.

Parameters

- **nTrials** (*int*) – Number of trials to run.
- **intensityVals** (*collection of floats*) – The complete set of possible stimulus levels. Note that the stimulus levels are not necessarily limited to intensities (as the name of this parameter implies), but they could also be contrasts, durations, weights, etc.
- **thresholdVals** (*float or collection of floats*) – The complete set of possible threshold values.
- **slopeVals** (*float or collection of floats*) – The complete set of possible slope values.
- **lowerAsymptoteVals** (*float or collection of floats*) – The complete set of possible values of the lower asymptote. This corresponds to false-alarm rates in yes-no tasks, and to the guessing rate in n-AFC tasks. Therefore, when performing an n-AFC experiment, the collection should consists of a single value only (e.g., *[0.5]* for 2-AFC, *[0.33]* for 3-AFC, *[0.25]* for 4-AFC, etc.).
- **lapseRateVals** (*float or collection of floats*) – The complete set of possible lapse rate values. The lapse rate defines the upper asymptote of the psychometric function, which will be at *1 - lapse rate*.
- **responseVals** (*collection*) – The complete set of possible response outcomes. Currently, only two outcomes are supported: the first element must correspond to a successful response / stimulus detection, and the second one to an unsuccessful or incorrect response. For example, in a yes-no task, one would use *['Yes', 'No']*, and in an n-AFC task, *['Correct', 'Incorrect']*; or, alternatively, the less verbose *[1, 0]* in both cases.

- **prior** (*dict of floats*) – The prior probabilities to assign to the parameter values. The dictionary keys correspond to the respective parameters: `threshold`, `slope`, `lowerAsymptote`, `lapseRate`.
- **startIntensity** (*float*) – The very first intensity (or stimulus level) to present.
- **psychometricFunc** (`{'weibull'}`) – The psychometric function to fit. Currently, only the Weibull function is supported.
- **stimScale** (`{'log10', 'dB', 'linear'}`) – The scale on which the stimulus intensities (or stimulus levels) are provided. Currently supported are the decadic logarithm, *log10*; decibels, *dB*; and a linear scale, *linear*.
- **stimSelectionMethod** (`{'minEntropy', 'minNEntropy'}`) – How to select the next stimulus. *minEntropy* will select the stimulus that will minimize the expected entropy. *minNEntropy* will randomly pick a stimulus from the set of stimuli that will produce the smallest, 2nd-smallest, ..., N-smallest entropy. This can be used to ensure some variation in the stimulus selection (and subsequent presentation) procedure. The number *N* will then have to be specified via the *stimSelectionOption* parameter.
- **stimSelectionOptions** (*dict*) – This parameter further controls how to select the next stimulus in case *stimSelectionMethod*=*minNEntropy*. The dictionary supports two keys: *N* and *maxConsecutiveReps*. *N* defines the number of “best” stimuli (i.e., those which produce the smallest *N* expected entropies) from which to randomly select a stimulus for presentation in the next trial. *maxConsecutiveReps* defines how many times the exact same stimulus can be presented on consecutive trials. For example, to randomly pick a stimulus from those which will produce the 4 smallest expected entropies, and to allow the same stimulus to be presented on two consecutive trials max, use *stimSelectionOptions*=`dict(N=4, maxConsecutiveReps=2)`. To achieve reproducible results, you may pass a seed to the random number generator via the *randomSeed* key.
- **paramEstimationMethod** (`{'mean', 'mode'}`) – How to calculate the final parameter estimate. *mean* returns the mean of each parameter, weighted by their respective posterior probabilities. *mode* returns the parameters at the peak of the posterior distribution.
- **extraInfo** (*dict*) – Additional information to store along the actual QUEST+ staircase data.
- **name** (*str*) – The name of the QUEST+ staircase object. This will appear in the PsychoPy logs.
- **label** (*str*) – Only used by *MultiStairHandler*, and otherwise ignored.
- **kwargs** (*dict*) – Additional keyword arguments. These might be passed, for example, through a *MultiStairHandler*, and will be ignored. A warning will be emitted whenever additional keyword arguments have been passed.

Warns **RuntimeWarning** – If an unknown keyword argument was passed.

Notes

The QUEST+ algorithm was first described by¹.

`_intensityDec()`

decrement the current intensity and reset counter

`_intensityInc()`

increment the current intensity and reset counter

`_terminate()`

Remove references to ourself in experiments and terminate the loop

`addData(result, intensity=None)`

Deprecated since 1.79.00: This function name was ambiguous. Please use one of these instead:

- `.addResponse(result, intensity)`
- `.addOtherData('dataName', value')`

`addOtherData(dataName, value)`

Add additional data to the handler, to be tracked alongside the result data but not affecting the value of the staircase

`addResponse(response, intensity=None)`

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial.

This is essential to advance the staircase to a new intensity level!

Supplying an *intensity* value here indicates that you did not use the recommended intensity in your last trial and the staircase will replace its recorded value with the one you supplied here.

`calculateNextIntensity()`

Based on current intensity, counter of correct responses, and current direction.

`getExp()`

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

`getOriginPathAndFile(originPath=None)`

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If *originPath* is provided (e.g. from Builder) then this is used otherwise the calling script is the *originPath* (fine from a standard python script).

property `intensity`

The intensity (level) of the current staircase

`next()`

Advances to next trial and returns it. Updates attributes; *thisTrial*, *thisTrialN* and *thisIndex*.

If the trials have ended, calling this method will raise a `StopIteration` error. This can be handled with code such as:

```
staircase = data.StairHandler(.....)
for eachTrial in staircase: # automatically stops when done
    # do stuff
```

or:

¹ Andrew B. Watson (2017). QUEST+: A general multidimensional Bayesian adaptive psychometric method. *Journal of Vision*, 17(3):10. doi: 10.1167/17.3.10.

```

staircase = data.StairHandler(.....)
while True: # ie forever
    try:
        thisTrial = staircase.next()
    except StopIteration: # we got a StopIteration error
        break # break out of the forever loop
# do stuff here for the trial

```

property paramEstimate

The estimated parameters of the psychometric function.

Returns A dictionary whose keys correspond to the names of the estimated parameters.

Return type dict of floats

property posterior

The marginal posterior distributions.

Returns A dictionary whose keys correspond to the names of the estimated parameters.

Return type dict of np.ndarrays

printAsText (*stimOut=None, dataOut='all_mean', 'all_std', 'all_raw', delim='\t', matrixOnly=False*)

Exactly like saveAsText() except that the output goes to the screen instead of a file

property prior

The marginal prior distributions.

Returns A dictionary whose keys correspond to the names of the parameters.

Return type dict of np.ndarrays

saveAsExcel (*fileName, sheetName='data', matrixOnly=False, appendFile=True, fileCollision-Method='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see *TrialHandler.saveAsText()*) that data can be stored in multiple named sheets within the file. So you could have a single file named after your experiment and then have one worksheet for each participant. Or you could have one file for each participant and then multiple sheets for repeated sessions etc.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level ('intensity') at each reversal, a list of reversal indices (trial numbers), the raw staircase / intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: string the name of the file to create or append. Can include relative or absolute path.

sheetName: string the name of the worksheet within the file

matrixOnly: True or False If set to True then only the data itself will be output (no additional info)

appendFile: True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: **string** Collision method passed to `handleFileCollision()`
This is ignored if `appendFile` is `True`.

saveAsJson (*fileName=None, encoding='utf-8-sig', fileCollisionMethod='rename'*)
Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.
- **encoding** (*string, optional*) – The encoding to use when writing the file.
- **fileCollisionMethod** (*string*) – Collision method passed to `handleFileCollision()`. Can be either of *'rename'*, *'overwrite'*, or *'fail'*.

Notes

Currently, a copy of the object is created, and the copy's `.origin` attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle (*fileName, fileCollisionMethod='rename'*)
Basically just saves a copy of self (with data) to a pickle file.

This can be reloaded if necessary and further analyses carried out.

Parameters **fileCollisionMethod:** Collision method passed to `handleFileCollision()`

saveAsText (*fileName, delim=None, matrixOnly=False, fileCollisionMethod='rename', encoding='utf-8-sig'*)
Write a text file with the data

Parameters

fileName: **a string** The name of the file, including path if needed. The extension `.tsv` will be added if not included.

delim: **a string** the delimiter to be used (e.g. `'\t'` for tab-delimited, `','` for csv files)

matrixOnly: **True/False** If `True`, prevents the output of the `extraInfo` provided at initialisation.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to `utf-8-sig`.

setExp (*exp*)
Sets the `ExperimentHandler` that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the `weakref` module.

property `startIntensity`

9.12.9 MultiStairHandler

```
class psychopy.data.MultiStairHandler (stairType='simple', method='random', conditions=None, nTrials=50, randomSeed=None, originPath=None, name='', autoLog=True)
```

A Handler to allow easy interleaved staircase procedures (simple or QUEST).

Parameters for the staircases, as used by the relevant *StairHandler* or *QuestHandler* (e.g. the *startVal*, *minVal*, *maxVal*...) should be specified in the *conditions* list and may vary between each staircase. In particular, the conditions **must** include a *startVal* (because this is a required argument to the above handlers), a *label* to tag the staircase and a *startValSd* (only for QUEST staircases). Any parameters not specified in the conditions file will revert to the default for that individual handler.

If you need to customize the behaviour further you may want to look at the recipe on *Coder - interleave staircases*.

Params

stairType: 'simple', 'quest', or 'questplus'

Use a *StairHandler*, a *QuestHandler*, or a *QuestPlusHandler*.

method: 'random', 'fullRandom', or 'sequential' If *random*, stairs are shuffled in each repeat but not randomized more than that (so you can't have 3 repeats of the same staircase in a row unless it's the only one still running). If *fullRandom*, the staircase order is "fully" randomized, meaning that, theoretically, a large number of subsequent trials could invoke the same staircase repeatedly. If *sequential*, don't perform any randomization.

conditions: a list of dictionaries specifying conditions Can be used to control parameters for the different staircases. Can be imported from an Excel file using *psychopy.data.importConditions* MUST include keys providing, 'startVal', 'label' and 'startValSd' (QUEST only). The 'label' will be used in data file saving so should be unique. See Example Usage below.

nTrials=50 Minimum trials to run (but may take more if the staircase hasn't also met its minimal reversals. See *StairHandler*

randomSeed [int or None] The seed with which to initialize the random number generator (RNG). If *None* (default), do not initialize the RNG with a specific value.

Example usage:

```
conditions=[
    {'label':'low', 'startVal': 0.1, 'ori':45},
    {'label':'high', 'startVal': 0.8, 'ori':45},
    {'label':'low', 'startVal': 0.1, 'ori':90},
    {'label':'high', 'startVal': 0.8, 'ori':90},
]
stairs = data.MultiStairHandler(conditions=conditions, nTrials=50)

for thisIntensity, thisCondition in stairs:
    thisOri = thisCondition['ori']

    # do something with thisIntensity and thisOri

    stairs.addResponse(correctIncorrect) # this is ESSENTIAL

# save data as multiple formats
stairs.saveDataAsExcel(fileName) # easy to browse
stairs.saveAsPickle(fileName) # contains more info
```

Raises **ValueError** – If an unknown randomization option was passed via the *method* keyword argument.

__startNewPass ()

Create a new iteration of the running staircases for this pass.

This is not normally needed by the user - it gets called at `__init__` and every time that `next()` runs out of trials for this pass.

__terminate ()

Remove references to ourself in experiments and terminate the loop

addData (*result, intensity=None*)

Deprecated 1.79.00: It was ambiguous whether you were adding the response (0 or 1) or some other data concerning the trial so there is now a pair of explicit methods:

- **addResponse(corr,intensity) #some data that alters the next** trial value
- **addOtherData('RT', reactionTime) #some other data that won't** control staircase

addOtherData (*name, value*)

Add some data about the current trial that will not be used to control the staircase(s) such as reaction time data

addResponse (*result, intensity=None*)

Add a 1 or 0 to signify a correct / detected or incorrect / missed trial

This is essential to advance the staircase to a new intensity level!

getExp ()

Return the ExperimentHandler that this handler is attached to, if any. Returns None if not attached

getOriginPathAndFile (*originPath=None*)

Attempts to determine the path of the script that created this data file and returns both the path to that script and its contents. Useful to store the entire experiment with the data.

If *originPath* is provided (e.g. from Builder) then this is used otherwise the calling script is the *originPath* (fine from a standard python script).

property intensity

The intensity (level) of the current staircase

next ()

Advances to next trial and returns it.

This can be handled with code such as:

```
staircase = data.MultiStairHandler(.....)
for eachTrial in staircase: # automatically stops when done
    # do stuff here for the trial
```

or:

```
staircase = data.MultiStairHandler(.....)
while True: # ie forever
    try:
        thisTrial = staircase.next()
    except StopIteration: # we got a StopIteration error
        break # break out of the forever loop
    # do stuff here for the trial
```


printAsText (*delim='\t', matrixOnly=False*)

Write the data to the standard output stream

Parameters

delim: a string the delimiter to be used (e.g. ‘\t’ for tab-delimited, ‘,’ for csv files)

matrixOnly: True/False If True, prevents the output of the *extraInfo* provided at initialisation.

saveAsExcel (*fileName, matrixOnly=False, appendFile=False, fileCollisionMethod='rename'*)

Save a summary data file in Excel OpenXML format workbook (*xlsx*) for processing in most spreadsheet packages. This format is compatible with versions of Excel (2007 or greater) and with OpenOffice (>=3.0).

It has the advantage over the simpler text files (see *TrialHandler.saveAsText()*) that the data from each staircase will be save in the same file, with the sheet name coming from the ‘label’ given in the dictionary of conditions during initialisation of the Handler.

The file extension *.xlsx* will be added if not given already.

The file will contain a set of values specifying the staircase level (‘intensity’) at each reversal, a list of reversal indices (trial numbers), the raw staircase/intensity level on *every* trial and the corresponding responses of the participant on every trial.

Parameters

fileName: string the name of the file to create or append. Can include relative or absolute path

matrixOnly: True or False If set to True then only the data itself will be output (no additional info)

appendFile: True or False If False any existing file with this name will be overwritten. If True then a new worksheet will be appended. If a worksheet already exists with that name a number will be added to make it unique.

fileCollisionMethod: string Collision method passed to *handleFileCollision()*
This is ignored if *append* is True.

saveAsJson (*fileName=None, encoding='utf-8-sig', fileCollisionMethod='rename'*)

Serialize the object to the JSON format.

Parameters

- **fileName** (*string, or None*) – the name of the file to create or append. Can include a relative or absolute path. If *None*, will not write to a file, but return an in-memory JSON object.

- **encoding** (*string, optional*) – The encoding to use when writing the file.

- **fileCollisionMethod** (*string*) – Collision method passed to *handleFileCollision()*. Can be either of ‘*rename*’, ‘*overwrite*’, or ‘*fail*’.

Notes

Currently, a copy of the object is created, and the copy's `.origin` attribute is set to an empty string before serializing because loading the created JSON file would sometimes fail otherwise.

saveAsPickle (*fileName*, *fileCollisionMethod*='rename')

Saves a copy of self (with data) to a pickle file.

This can be reloaded later and further analyses carried out.

Parameters `fileCollisionMethod`: Collision method passed to `handleFileCollision()`

saveAsText (*fileName*, *delim*=None, *matrixOnly*=False, *fileCollisionMethod*='rename', *encoding*='utf-8-sig')

Write out text files with the data.

For `MultiStairHandler` this will output one file for each staircase that was run, with `_label` added to the `fileName` that you specify above (label comes from the condition dictionary you specified when you created the Handler).

Parameters

fileName: a string The name of the file, including path if needed. The extension `.tsv` will be added if not included.

delim: a string the delimiter to be used (e.g. `' '` for tab-delimited, `' '` for csv files)

matrixOnly: True/False If True, prevents the output of the `extraInfo` provided at initialisation.

fileCollisionMethod: Collision method passed to `handleFileCollision()`

encoding: The encoding to use when saving a the file. Defaults to `utf-8-sig`.

setExp (*exp*)

Sets the `ExperimentHandler` that this handler is attached to

Do NOT attempt to set the experiment using:

```
trials._exp = myExperiment
```

because it needs to be performed using the `weakref` module.

9.12.10 FitWeibull

class `psychopy.data.FitWeibull` (*xx*, *yy*, *sems*=1.0, *guess*=None, *display*=1, *expectedMin*=0.5, *optimize_kws*=None)

Fit a Weibull function (either 2AFC or YN) of the form:

```
y = chance + (1.0-chance) * (1-exp( -(xx/alpha)**(beta) ))
```

and with inverse:

```
x = alpha * (-log((1.0-y)/(1-chance)))**(1.0/beta)
```

After fitting the function you can evaluate an array of `x`-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with `[alpha, beta]`)

_doFit ()

The Fit class that derives this needs to specify its `_evalFunction`

eval (*xx, params=None*)

Evaluate *xx* for the current parameters of the model, or for arbitrary params if these are given.

inverse (*yy, params=None*)

Evaluate *yy* for the current parameters of the model, or for arbitrary params if these are given.

9.12.11 FitLogistic

class psychopy.data.**FitLogistic** (*xx, yy, sems=1.0, guess=None, display=1, expectedMin=0.5, optimize_kws=None*)

Fit a Logistic function (either 2AFC or YN) of the form:

$$y = \text{chance} + (1-\text{chance}) / (1 + \exp((\text{PSE}-xx) * \text{JND}))$$

and with inverse:

$$x = \text{PSE} - \log((1-\text{chance}) / (yy-\text{chance}) - 1) / \text{JND}$$

After fitting the function you can evaluate an array of *x*-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with [PSE, JND])

_doFit ()

The Fit class that derives this needs to specify its `_evalFunction`

eval (*xx, params=None*)

Evaluate *xx* for the current parameters of the model, or for arbitrary params if these are given.

inverse (*yy, params=None*)

Evaluate *yy* for the current parameters of the model, or for arbitrary params if these are given.

9.12.12 FitNakaRushton

class psychopy.data.**FitNakaRushton** (*xx, yy, sems=1.0, guess=None, display=1, expectedMin=0.5, optimize_kws=None*)

Fit a Naka-Rushton function of the form:

$$yy = rMin + (rMax-rMin) * xx**n / (xx**n + c50**n)$$

After fitting the function you can evaluate an array of *x*-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with [rMin, rMax, c50, n])

Note that this differs from most of the other functions in not using a value for the expected minimum. Rather, it fits this as one of the parameters of the model.

_doFit ()

The Fit class that derives this needs to specify its `_evalFunction`

eval (*xx, params=None*)

Evaluate *xx* for the current parameters of the model, or for arbitrary params if these are given.

inverse (*yy, params=None*)

Evaluate *yy* for the current parameters of the model, or for arbitrary params if these are given.

9.12.13 FitCumNormal

`class psychopy.data.FitCumNormal (xx, yy, sems=1.0, guess=None, display=1, expectedMin=0.5, optimize_kws=None)`

Fit a Cumulative Normal function (aka error function or erf) of the form:

```
y = chance + (1-chance) * ((special.erf((xx-xShift)/(sqrt(2)*sd))+1)*0.5)
```

and with inverse:

```
x = xShift+sqrt(2)*sd*(erfinv(((yy-chance)/(1-chance)-.5)*2))
```

After fitting the function you can evaluate an array of x-values with `fit.eval(x)`, retrieve the inverse of the function with `fit.inverse(y)` or retrieve the parameters from `fit.params` (a list with [centre, sd] for the Gaussian distribution forming the cumulative)

NB: Prior to version 1.74 the parameters had different meaning, relating to `xShift` and slope of the function (similar to `1/sd`). Although that is more in with the parameters for the Weibull fit, for instance, it is less in keeping with standard expectations of normal (Gaussian distributions) so in version 1.74.00 the parameters became the [centre,sd] of the normal distribution.

`_doFit ()`

The Fit class that derives this needs to specify its `_evalFunction`

`eval (xx, params=None)`

Evaluate `xx` for the current parameters of the model, or for arbitrary params if these are given.

`inverse (yy, params=None)`

Evaluate `yy` for the current parameters of the model, or for arbitrary params if these are given.

9.12.14 importConditions ()

`psychopy.data.importConditions (fileName, returnFieldNames=False, selection="")`

Imports a list of conditions from an .xlsx, .csv, or .pkl file

The output is suitable as an input to `TrialHandler trialList` or to `MultiStairHandler` as a `conditions` list.

If `fileName` ends with:

- **.csv: import as a comma-separated-value file** (header + row x col)
- **.xlsx: import as Excel 2007 (xlsx) files.** No support for older (.xls) is planned.
- **.pkl: import from a pickle file as list of lists** (header + row x col)

The file should contain one row per type of trial needed and one column for each parameter that defines the trial type. The first row should give parameter names, which should:

- be unique
- begin with a letter (upper or lower case)
- contain no spaces or other punctuation (underscores are permitted)

`selection` is used to select a subset of condition indices to be used It can be a list/array of indices, a python `slice` object or a string to be parsed as either option. e.g.:

- “1,2,4” or [1,2,4] or (1,2,4) are the same
- “2:5” # 2, 3, 4 (doesn’t include last whole value)

- “-10:2:” # tenth from last to the last in steps of 2
- slice(-10, 2, None) # the same as above
- random(5) * 8 # five random vals 0-7

9.12.15 functionFromStaircase ()

psychopy.data.**functionFromStaircase** (*intensities, responses, bins=10*)

Create a psychometric function by binning data from a staircase procedure. Although the default is 10 bins Jon now always uses ‘unique’ bins (fewer bins looks pretty but leads to errors in slope estimation)

usage:

```
intensity, meanCorrect, n = functionFromStaircase(intensities,
                                                responses, bins)
```

where:

intensities are a list (or array) of intensities to be binned

responses are a list of 0,1 each corresponding to the equivalent intensity value

bins can be an integer (giving that number of bins) or ‘unique’ (each bin is made from aa data for exactly one intensity value)

intensity a numpy array of intensity values (where each is the center of an intensity bin)

meanCorrect a numpy array of mean % correct in each bin

n a numpy array of number of responses contributing to each mean

9.12.16 bootStraps ()

psychopy.data.**bootStraps** (*dat, n=1*)

Create a list of n bootstrapped resamples of the data

SLOW IMPLEMENTATION (Python for-loop)

Usage: out = bootStraps(dat, n=1)

Where:

dat an NxM or 1xN array (each row is a different condition, each column is a different trial)

n number of bootstrapped resamples to create

out

- dim[0]=conditions
- dim[1]=trials
- dim[2]=resamples

9.13 Encryption

Some labs may wish to better protect their data from casual inspection or accidental disclosure. This is possible within using a separate python package, pyFileSec, which grew out of . pyFileSec is distributed with the StandAlone versions of , or can be installed using pip or easy_install via <https://pypi.python.org/pypi/PyFileSec/>

Some elaboration of pyFileSec usage and security strategy can be found here: <https://pythonhosted.org/PyFileSec>

Basic usage is illustrated in the Coder demo > misc > encrypt_data.py

9.14 psychopy.event - for keypresses and mouse clicks

class psychopy.event.**Mouse** (*visible=True, newPos=None, win=None*)

Easy way to track what your mouse is doing.

It needn't be a class, but since Joystick works better as a class this may as well be one too for consistency

Create your *visual.Window* before creating a Mouse.

Parameters

visible [**True** or **False**] makes the mouse invisible if necessary

newPos [**None** or [x,y]] gives the mouse a particular starting position (pygame *Window* only)

win [**None** or *Window*] the window to which this mouse is attached (the first found if **None** provided)

clickReset (*buttons=0, 1, 2*)

Reset a 3-item list of core.Clocks use in timing button clicks.

The pyglet mouse-button-pressed handler uses their clock.getLastResetTime() when a button is pressed so the user can reset them at stimulus onset or offset to measure RT. The default is to reset all, but they can be reset individually as specified in buttons list

getPos ()

Returns the current position of the mouse, in the same units as the *Window* (0,0) is at centre

getPressed (*getTime=False*)

Returns a 3-item list indicating whether or not buttons 0,1,2 are currently pressed.

If *getTime=True* (False by default) then *getPressed* will return all buttons that have been pressed since the last call to *mouse.clickReset* as well as their time stamps:

```
buttons = mouse.getPressed()
buttons, times = mouse.getPressed(getTime=True)
```

Typically you want to call *mouse.clickReset*() at stimulus onset, then after the button is pressed in reaction to it, the total time elapsed from the last reset to click is in *mouseTimes*. This is the actual RT, regardless of when the call to *getPressed*() was made.

getRel ()

Returns the new position of the mouse relative to the last call to *getRel* or *getPos*, in the same units as the *Window*.

getVisible ()

Gets the visibility of the mouse (1 or 0)

getWheelRel ()

Returns the travel of the mouse scroll wheel since last call. Returns a numpy.array(x,y) but for most wheels y is the only value that will change (except Mac mighty mice?)

isPressedIn (shape, buttons=0, 1, 2)

Returns *True* if the mouse is currently inside the shape and one of the mouse buttons is pressed. The default is that any of the 3 buttons can indicate a click; for only a left-click, specify *buttons=[0]*:

```
if mouse.isPressedIn(shape) :
if mouse.isPressedIn(shape, buttons=[0]) : # left-clicks only
```

Ideally, *shape* can be anything that has a *.contains()* method, like *ShapeStim* or *Polygon*. Not tested with *ImageStim*.

mouseMoveTime ()

mouseMoved (distance=None, reset=False)

Determine whether/how far the mouse has moved.

With no args returns true if mouse has moved at all since last *getPos()* call, or distance (x,y) can be set to pos or neg distances from x and y to see if moved either x or y that far from lastPos, or distance can be an int/float to test if new coordinates are more than that far in a straight line from old coords.

Retrieve time of last movement from *self.mouseClock.getTime()*.

Reset can be to 'here' or to screen coords (x,y) which allows measuring distance from there to mouse when moved. If reset is (x,y) and distance is set, then *prevPos* is set to (x,y) and distance from (x,y) to here is checked, *mouse.lastPos* is set as current (x,y) by *getPos()*, *mouse.prevPos* holds lastPos from last time *mouseMoved* was called.

setExclusive (exclusivity)

Binds the mouse to the experiment window. Only works in Pyglet.

In multi-monitor settings, or with a window that is not fullscreen, the mouse pointer can drift, and thereby PsychoPy might not get the events from that window. *setExclusive(True)* works with Pyglet to bind the mouse to the experiment window.

Note that binding the mouse pointer to a window will cause the pointer to vanish, and absolute positions will no longer be meaningful *getPos()* returns [0, 0] in this case.

setPos (newPos=0, 0)

Sets the current position of the mouse, in the same units as the *Window*. (0,0) is the center.

Parameters

newPos [(x,y) or [x,y]] the new position on the screen

setVisible (visible)

Sets the visibility of the mouse to 1 or 0

NB when the mouse is not visible its absolute position is held at (0, 0) to prevent it from going off the screen and getting lost! You can still use *getRel()* in that case.

property units

The units for this mouse (will match the current units for the *Window* it lives in)

psychopy.event.clearEvents (eventType=None)

Clears all events currently in the event buffer.

Optional argument, *eventType*, specifies only certain types to be cleared.

Parameters

eventType [**None**, 'mouse', 'joystick', 'keyboard'] If this is not None then only events of the given type are cleared

`psychoPy.event.waitKeys` (*maxWait=inf*, *keyList=None*, *modifiers=False*, *timeStamped=False*, *clearEvents=True*)

Same as `~psychoPy.event.getKeys`, but halts everything (including drawing) while awaiting input from keyboard.

Parameters

maxWait [any numeric value.] Maximum number of seconds period and which keys to wait for. Default is float('inf') which simply waits forever.

keyList [**None** or []] Allows the user to specify a set of keys to check for. Only keypresses from this set of keys will be removed from the keyboard buffer. If the *keyList* is *None*, all keys will be checked and the key buffer will be cleared completely. NB, pygame doesn't return timestamps (they are always 0)

modifiers [**False** or True] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, modifiers). The modifiers are a dict of keyboard modifier flags keyed by the modifier name (eg. 'shift', 'ctrl').

timeStamped [**False**, True, or *Clock*] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, time). If a *core.Clock* is given then the time will be relative to the *Clock*'s last reset.

clearEvents [**True** or False] Whether to clear the keyboard event buffer (and discard preceding keypresses) before starting to monitor for new keypresses.

Returns None if times out.

`psychoPy.event.getKeys` (*keyList=None*, *modifiers=False*, *timeStamped=False*)

Returns a list of keys that were pressed.

Parameters

keyList [**None** or []] Allows the user to specify a set of keys to check for. Only keypresses from this set of keys will be removed from the keyboard buffer. If the *keyList* is *None*, all keys will be checked and the key buffer will be cleared completely. NB, pygame doesn't return timestamps (they are always 0)

modifiers [**False** or True] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, modifiers). The modifiers are a dict of keyboard modifier flags keyed by the modifier name (eg. 'shift', 'ctrl').

timeStamped [**False**, True, or *Clock*] If True will return a list of tuples instead of a list of keynames. Each tuple has (keyname, time). If a *core.Clock* is given then the time will be relative to the *Clock*'s last reset.

Author

- 2003 written by Jon Peirce
- 2009 keyList functionality added by Gary Strangman
- 2009 timeStamped code provided by Dave Britton
- 2016 modifiers code provided by 5AM Solutions

`psychoPy.event.xydist` (*p1=0.0, 0.0*, *p2=0.0, 0.0*)

Helper function returning the cartesian distance between p1 and p2

9.15 psychopy.filters - helper functions for creating filters

This module has moved to `psychopy.visual.filters` but you can still (currently) import it as `psychopy.filters`

Various useful functions for creating filters and textures (e.g. for PatchStim)

`psychopy.visual.filters.butter2d_bp` (*size, cutin, cutoff, n*)

Bandpass Butterworth filter in two dimensions.

Parameters

size [tuple] size of the filter

cutin [float] relative cutin frequency of the filter (0 - 1.0)

cutoff [float] relative cutoff frequency of the filter (0 - 1.0)

n [int, optional] order of the filter, the higher n is the sharper the transition is.

Returns

numpy.ndarray filter kernel in 2D centered

`psychopy.visual.filters.butter2d_hp` (*size, cutoff, n=3*)

Highpass Butterworth filter in two dimensions.

Parameters

size [tuple] size of the filter

cutoff [float] relative cutoff frequency of the filter (0 - 1.0)

n [int, optional] order of the filter, the higher n is the sharper the transition is.

Returns

numpy.ndarray: filter kernel in 2D centered

`psychopy.visual.filters.butter2d_lp` (*size, cutoff, n=3*)

Create lowpass 2D Butterworth filter.

Parameters

size [tuple] size of the filter

cutoff [float] relative cutoff frequency of the filter (0 - 1.0)

n [int, optional] order of the filter, the higher n is the sharper the transition is.

Returns

numpy.ndarray filter kernel in 2D centered

`psychopy.visual.filters.butter2d_lp_elliptic` (*size, cutoff_x, cutoff_y, n=3, alpha=0, offset_x=0, offset_y=0*)

Butterworth lowpass filter of any elliptical shape.

Parameters

size [tuple] size of the filter

cutoff_x, cutoff_y [float, float] relative cutoff frequency of the filter (0 - 1.0) for x and y axes

alpha [float, optional] rotation angle (in radians)

offset_x, offset_y [float] offsets for the ellipsoid

n [int, optional] order of the filter, the higher n is the sharper the transition is.

Returns

numpy.ndarray: filter kernel in 2D centered

`psychoPy.visual.filters.conv2d(smaller, larger)`
Convolve a pair of 2d numpy matrices.

Uses fourier transform method, so faster if larger matrix has dimensions of size $2^{**}n$

Actually right now the matrices must be the same size (will sort out padding issues another day!)

`psychoPy.visual.filters.getRMScontrast(matrix)`
Returns the RMS contrast (the sample standard deviation) of a array

`psychoPy.visual.filters.imfft(X)`
Perform 2D FFT on an image and center low frequencies

`psychoPy.visual.filters.imiFFT(X)`
Inverse 2D FFT with decentering

`psychoPy.visual.filters.make2DGauss(x, y, mean=0.0, sd=1.0, gain=1.0, base=0.0)`
Return the gaussian distribution for a given set of x-vals

Parameters

- **x** – should be x and y indexes as might be created by `numpy.mgrid`
- **y** – should be x and y indexes as might be created by `numpy.mgrid`
- **mean** (*float*) – the centre of the distribution - may be a tuple
- **sd** (*float*) – the width of the distribution - may be a tuple
- **gain** (*float*) – the height of the distribution
- **base** (*float*) – an offset added to the result

`psychoPy.visual.filters.makeGauss(x, mean=0.0, sd=1.0, gain=1.0, base=0.0)`
Return the gaussian distribution for a given set of x-vals

Parameters

mean: float the centre of the distribution
sd: float the width of the distribution
gain: float the height of the distribution
base: float an offset added to the result

`psychoPy.visual.filters.makeGrating(res, ori=0.0, cycles=1.0, phase=0.0, gratType='sin',
contr=1.0)`
Make an array containing a luminance grating of the specified params

Parameters

res: integer the size of the resulting matrix on both dimensions (e.g 256)
ori: float or int (default=0.0) the orientation of the grating in degrees
cycles: float or int (default=1.0) the number of grating cycles within the array
phase: float or int (default=0.0) the phase of the grating in degrees (NB this differs to most PsychoPy phase arguments which use units of fraction of a cycle)

gratType: 'sin', 'sqr', 'ramp' or 'sinXsin' (default="sin") the type of grating to be 'drawn'

contr: float (default=1.0) contrast of the grating

Returns a square numpy array of size resXres

`psychopy.visual.filters.makeMask` (*matrixSize*, *shape='circle'*, *radius=1.0*, *center=0.0, 0.0*, *range=-1, 1*, *fringeWidth=0.2*)

Returns a matrix to be used as an alpha mask (circle,gauss,ramp).

Parameters

matrixSize: integer the size of the resulting matrix on both dimensions (e.g 256)

shape: 'circle','gauss','ramp' (linear gradient from center), 'raisedCosine' (the edges are blurred by a raised cosine) shape of the mask

radius: float scale factor to be applied to the mask (circle with radius of [1,1] will extend just to the edge of the matrix). Radius can asymmetric, e.g. [1.0,2.0] will be wider than it is tall.

center: 2x1 tuple or list (default=[0.0,0.0]) the centre of the mask in the matrix ([1,1] is top-right corner, [-1,-1] is bottom-left)

fringeWidth: float (0-1) The proportion of the raisedCosine that is being blurred.

range: 2x1 tuple or list (default=[-1,1]) The minimum and maximum value in the mask matrix

`psychopy.visual.filters.makeRadialMatrix` (*matrixSize*, *center=0.0, 0.0*, *radius=1.0*)

Generate a square matrix where each element values is its distance from the centre of the matrix.

Parameters

- **matrixSize** (*int*) – Matrix size. Corresponds to the number of elements along each dimension. Must be >0.
- **radius** (*float*) – scale factor to be applied to the mask (circle with radius of [1,1] will extend just to the edge of the matrix). Radius can be asymmetric, e.g. [1.0,2.0] will be wider than it is tall.
- **center** (*2x1 tuple or list (default=[0.0, 0.0])*) – the centre of the mask in the matrix ([1,1] is top-right corner, [-1,-1] is bottom-left)

Returns Square matrix populated with distance values and *size == (matrixSize, matrixSize)*.

Return type ndarray

`psychopy.visual.filters.maskMatrix` (*matrix*, *shape='circle'*, *radius=1.0*, *center=0.0, 0.0*)

Make and apply a mask to an input matrix (e.g. a grating)

Parameters

matrix: a square numpy array array to which the mask should be applied

shape: 'circle','gauss','ramp' (linear gradient from center) shape of the mask

radius: float scale factor to be applied to the mask (circle with radius of [1,1] will extend just to the edge of the matrix). Radius can be asymmetric, e.g. [1.0,2.0] will be wider than it is tall.

center: 2x1 tuple or list (default=[0.0,0.0]) the centre of the mask in the matrix ([1,1] is top-right corner, [-1,-1] is bottom-left)

9.16 psychopy.gui - create dialogue boxes

9.16.1 DlgFromDict

```
class psychopy.gui.DlgFromDict (dictionary, title="", fixed=None, order=None, tip=None,  
                                screen=- 1, sortKeys=True, copyDict=False, labels=None,  
                                show=True, sort_keys=None, copy_dict=None)
```

Creates a dialogue box that represents a dictionary of values. Any values changed by the user are change (in-place) by this dialogue box.

Parameters

- **dictionary** (*dict*) – A dictionary defining the input fields (keys) and pre-filled values (values) for the user dialog
- **title** (*str*) – The title of the dialog window
- **labels** (*dict*) – A dictionary defining labels (values) to be displayed instead of key strings (keys) defined in *dictionary*. Not all keys in *dictionary* need to be contained in labels.
- **fixed** (*list*) – A list of keys for which the values shall be displayed in non-editable fields
- **order** (*list*) – A list of keys defining the display order of keys in *dictionary*. If not all keys in *dictionary* are contained in *order*, those will appear in random order after all ordered keys.
- **tip** (*list*) – A dictionary assigning tooltips to the keys
- **screen** (*int*) – Screen number where the Dialog is displayed. If -1, the Dialog will be displayed on the primary screen.
- **sortKeys** (*bool*) – A boolean flag indicating that keys are to be sorted alphabetically.
- **copyDict** (*bool*) – If False, modify *dictionary* in-place. If True, a copy of the dictionary is created, and the altered version (after user interaction) can be retrieved from `:attr:`~psychopy.gui.DlgFromDict.dictionary``.
- **labels** – A dictionary defining labels (dict values) to be displayed instead of key strings (dict keys) defined in *dictionary*. Not all keys in *dictionary* need to be contained in labels.
- **show** (*bool*) – Whether to immediately display the dialog upon instantiation. If False, it can be displayed at a later time by calling its *show()* method.
- **e.g.** –
- **::** –

```
info = {'Observer': 'jwp', 'GratingOri': 45, 'ExpVersion': 1.1, 'Group': ['Test', 'Control']}  
infoDlg = gui.DlgFromDict(dictionary=info, title='TestExperiment',  
                           fixed=['ExpVersion'])  
if infoDlg.OK: print(info)  
else: print('User Cancelled')
```
- **the code above** (*In*) –
- **contents of info will be updated to the values** (*the*) –
- **by the dialogue box.** (*returned*) –

- **the user cancels (rather than pressing OK) (If) –**

:param : :param then the dictionary remains unchanged. If you want to check whether: :param the user hit OK: :param then check whether DlgFromDict.OK equals: :param True or False: :param See GUI.py for a usage demo: :param including order and tip (tooltip):

show()

Display the dialog.

9.16.2 Dlg

class psychopy.gui.Dlg (title='PsychoPy Dialog', pos=None, size=None, style=None, labelButtonOK='OK', labelButtonCancel='Cancel', screen=-1)

A simple dialogue box. You can add text or input boxes (sequentially) and then retrieve the values.

see also the function *dlgFromDict* for an **even simpler** version

Example

```
from psychopy import gui

myDlg = gui.Dlg(title="JWP's experiment")
myDlg.addText('Subject info')
myDlg.addField('Name:')
myDlg.addField('Age:', 21)
myDlg.addText('Experiment Info')
myDlg.addField('Grating Ori:', 45)
myDlg.addField('Group:', choices=["Test", "Control"])
ok_data = myDlg.show() # show dialog and wait for OK or Cancel
if myDlg.OK: # or if ok_data is not None
    print(ok_data)
else:
    print('user cancelled')
```

addField (label="", initial="", color="", choices=None, tip="", enabled=True)

Adds a (labelled) input field to the dialogue box, optional text color and tooltip.

If 'initial' is a bool, a checkbox will be created. If 'choices' is a list or tuple, a dropdown selector is created. Otherwise, a text line entry box is created.

Returns a handle to the field (but not to the label).

addFixedField (label="", initial="", color="", choices=None, tip="")

Adds a field to the dialog box (like `addField`) but the field cannot be edited. e.g. Display experiment version.

show()

Presents the dialog and waits for the user to press OK or CANCEL.

If user presses OK button, function returns a list containing the updated values coming from each of the input fields created. Otherwise, None is returned.

Returns self.data

9.16.3 fileOpenDlg()

`psychoPy.gui.fileOpenDlg` (*tryFilePath=""*, *tryFileName=""*, *prompt='Select file to open'*, *allowed=None*)

A simple dialogue allowing read access to the file system.

Parameters

tryFilePath: string default file path on which to open the dialog

tryFileName: string default file name, as suggested file

prompt: string (default “Select file to open”) can be set to custom prompts

allowed: string (available since v1.62.01) a string to specify file filters. e.g. “Text files (*.txt) ;; Image files (*.bmp *.gif)” See <https://www.riverbankcomputing.com/static/Docs/PyQt4/qfiledialog.html#getOpenFileNames> for further details

If `tryFilePath` or `tryFileName` are empty or invalid then current path and empty names are used to start search.

If user cancels, then `None` is returned.

9.16.4 fileSaveDlg()

`psychoPy.gui.fileSaveDlg` (*initFilePath=""*, *initFileName=""*, *prompt='Select file to save'*, *allowed=None*)

A simple dialogue allowing write access to the file system. (Useful in case you collect an hour of data and then try to save to a non-existent directory!!)

Parameters

initFilePath: string default file path on which to open the dialog

initFileName: string default file name, as suggested file

prompt: string (default “Select file to open”) can be set to custom prompts

allowed: string a string to specify file filters. e.g. “Text files (*.txt) ;; Image files (*.bmp *.gif)” See <https://www.riverbankcomputing.com/static/Docs/PyQt4/qfiledialog.html#getSaveFileName> for further details

If `initFilePath` or `initFileName` are empty or invalid then current path and empty names are used to start search.

If user cancels the `None` is returned.

9.17 psychoPy.info - functions for getting information about the system

This module has tools for fetching data about the system or the current Python process. Such info can be useful for understanding the context in which an experiment was run.

class `psychoPy.info.RuntimeInfo` (*author=None*, *version=None*, *win=None*, *refreshTest='grating'*, *userProcsDetailed=False*, *verbose=False*)

Returns a snapshot of your configuration at run-time, for immediate or archival use.

Returns a dict-like object with info about PsychoPy, your experiment script, the system & OS, your window and monitor settings (if any), python & packages, and OpenGL.

If you want to skip testing the refresh rate, use `refreshTest=None`

Example usage: see `runtimeInfo.py` in coder demos.

Parameters

- **win** (*Window*, *False* or *None*) – What window to use for refresh rate testing (if any) and settings. *None* -> temporary window using defaults; *False* -> no window created, used, nor profiled; a *Window()* instance you have already created one.
- **author** (*str* or *None*) – *None* will try to autodetect first `__author__` in `sys.argv[0]`, whereas a *str* being user-supplied author info (of an experiment).
- **version** (*str* or *None*) – *None* try to autodetect first `__version__` in `sys.argv[0]` or *str* being the user-supplied version info (of an experiment).
- **verbose** (*bool*) – Show additional information. Default is *False*.
- **refreshTest** (*str*, *bool* or *None*) – True or ‘grating’ = assess refresh average, median, and SD of 60 `win.flip()`s, using `visual.getMsPerFrame()` ‘grating’ = show a visual during the assessment; *True* = assess without a visual. Default is ‘grating’.
- **userProcsDetailed** (*bool*) – Get details about concurrent user’s processes (command, process-ID). Default is *False*.

Returns

- A flat dict (but with several groups based on key names)
- **psychopy** (*version*, *rush()* availability) – `psychopyVersion`, `psychopyHaveExtRush`, git branch and current commit hash if available
- **experiment** (*author*, *version*, *directory*, *name*, *current time-stamp*, *SHA1*) – digest, VCS info (if any, svn or hg only), `experimentAuthor`, `experimentVersion`, ...
- **system** (*hostname*, *platform*, *user login*, *count of users*,) – user process info (count, cmd + pid), flagged processes `systemHostname`, `systemPlatform`, ...
- **window** ((*see output*; many details about the refresh rate, window,) – and monitor; units are noted) `windowWinType`, `windowWaitBlanking`, ... `windowRefreshTimeSD_ms`, ... `windowMonitor.<details>`, ...
- **python** (*version of python*, *versions of key packages*) – (wx, numpy, scipy, matplotlib, pygame, pygame) `pythonVersion`, `pythonScipyVersion`, ...
- **openGL** (*version*, *vendor*, *rendering engine*, *plus info on whether*) – several extensions are present `openGLVersion`, ..., `openGLExtGL_EXT_framebuffer_object`, ...

`__setCurrentProcessInfo` (*verbose=False*, *userProcsDetailed=False*)

What other processes are currently active for this user?

`__setExperimentInfo` (*author*, *version*, *verbose*)

Auto-detect `__author__` and `__version__` in `sys.argv[0]` (= the # users’s script)

`__setPythonInfo` ()

External python packages, python details

`__setSystemInfo` ()

System info

`__setWindowInfo` (*win*, *verbose=False*, *refreshTest='grating'*, *usingTempWin=True*)

Find and store info about the window: refresh rate, configuration info.

`psychopy.info.__getHgVersion` (*filename*)

Tries to discover the mercurial (hg) parent and id of a file.

Not thoroughly tested; untested on Windows Vista, Win 7, FreeBSD

Author

- 2010 written by Jeremy Gray

`psychoPy.info._getSha1hexDigest` (*thing*, *isfile=False*)

Returns base64 / hex encoded sha1 digest of `str(thing)`, or of a file contents. Return None if a file is requested but no such file exists

Author

- 2010 Jeremy Gray; updated 2011 to be more explicit,
- 2012 to remove `sha.new()`

```
>>> _getSha1hexDigest('1')
'356a192b7913b04c54574d18c28d46e6395428ab'
>>> _getSha1hexDigest(1)
'356a192b7913b04c54574d18c28d46e6395428ab'
```

`psychoPy.info._getSvnVersion` (*filename*)

Tries to discover the svn version (revision #) for a file.

Not thoroughly tested; untested on Windows Vista, Win 7, FreeBSD

Author

- 2010 written by Jeremy Gray

`psychoPy.info._getUserNameUID` ()

Return user name, UID.

UID values can be used to infer admin-level: -1=undefined, 0=full admin/root, >499=assume non-admin/root (>999 on debian-based)

Author

- 2010 written by Jeremy Gray

`psychoPy.info.getMemoryUsage` ()

Get the memory (RAM) currently used by this Python process, in M.

`psychoPy.info.getRAM` ()

Return system's physical RAM & available RAM, in M.

9.18 `psychoPy.layout` - For working with vectors and points

Classes and functions for working with coordinates systems.

9.18.1 Overview

<code>Vector</code> (value, units, win)	Class representing a vector.
<code>Position</code> (value, units[, win])	Class representing a position vector.
<code>Size</code> (value, units[, win])	Class representing a size.
<code>Vertices</code> (verts[, obj, size, pos, units, ...])	Class representing an array of vertices.

9.18.2 Details

class `psychopy.layout.Vector` (*value, units, win*)

Class representing a vector.

A vector is a mathematical construct that specifies a length (or magnitude) and direction within a given coordinate system. This class provides methods to manipulate vectors and convert them between coordinate systems.

This class may be used to assist in positioning stimuli on a screen.

Parameters

- **value** (*ArrayLike*) – Array of vector lengths along each dimension of the space the vector is within. Vectors are specified as either 1xN for single vectors, and Nx2 or Nx3 for multiple vectors.
- **units** (*str or None*) – Units which *value* has been specified in. Applicable values are *'pix'*, *'deg'*, *'degFlat'*, *'degFlatPos'*, *'cm'*, *'pt'*, *'norm'*, *'height'*, or *None*.
- **win** (*~psychopy.visual.Window or None*) – Window associated with this vector. This value must be specified if you wish to map vectors to coordinate systems that require additional information about the monitor the window is being displayed on.

Examples

Create a new vector object using coordinates specified in pixel (*'pix'*) units:

```
my_vector = Vector([256, 256], 'pix')
```

Multiple vectors may be specified by supplying a list of vectors:

```
my_vector = Vector([[256, 256], [640, 480]], 'pix')
```

Operators can be used to compare the magnitudes of vectors:

```
mag_is_same = vec1 == vec2 # same magnitude
mag_is_greater = vec1 > vec2 # one greater than the other
```

1xN vectors return a boolean value while Nx2 or Nx3 arrays return N-length arrays of boolean values.

property **cm**

Values in units of *'cm'* (centimeters).

property **copy()**

Create a copy of this object

property **deg**

Values in units of *'deg'* (degrees of visual angle).

property **degFlat**

Values in units of *'degFlat'* (degrees of visual angle corrected for screen curvature).

When dealing with positions/sizes in isolation; *'deg'*, *'degFlat'* and *'degFlatPos'* are synonymous - as the conversion is done at the vertex level.

property **degFlatPos**

Values in units of *'degFlatPos'*.

When dealing with positions/sizes in isolation; *'deg'*, *'degFlat'* and *'degFlatPos'* are synonymous - as the conversion is done at the vertex level.

property dimensions

How many dimensions (x, y, z) are specified?

property direction

Direction of vector (i.e. angle between vector and the horizontal plane).

property height

Value in units of 'height' (normalized to the height of the window).

property magnitude

Magnitude of vector (i.e. length of the line from vector to (0, 0) in pixels).

property monitor

The monitor used for calculations within this object (*~psychopy.monitors.Monitor*).

property norm

Value in units of 'norm' (normalized device coordinates).

property pix

Values in units of 'pix' (pixels).

property pt

Vector coordinates in 'pt' (points).

Points are commonly used in print media to define text sizes. One point is equivalent to 1/72 inches, or around 0.35 mm.

set (*value, units, win=None*)

validate (*value, units*)

Validate input values.

Ensures the values are in the correct format.

Returns Parameters *value* and *units*.

Return type `tuple`

class `psychopy.layout.Position` (*value, units, win=None*)

Class representing a position vector.

This class is used to specify the location of a point within some coordinate system (e.g., (x, y)).

Parameters

- **value** (*ArrayLike*) – Array of coordinates representing positions within a coordinate system. Positions are specified in a similar manner to *~psychopy.layout.Vector* as either 1xN for single vectors, and Nx2 or Nx3 for multiple positions.
- **units** (*str or None*) – Units which *value* has been specified in. Applicable values are 'pix', 'deg', 'degFlat', 'degFlatPos', 'cm', 'pt', 'norm', 'height', or *None*.
- **win** (*~psychopy.visual.Window or None*) – Window associated with this position. This value must be specified if you wish to map positions to coordinate systems that require additional information about the monitor the window is being displayed on.

property cm

Values in units of 'cm' (centimeters).

copy ()

Create a copy of this object

property deg

Values in units of 'deg' (degrees of visual angle).

property degFlat

Values in units of 'degFlat' (degrees of visual angle corrected for screen curvature).

When dealing with positions/sizes in isolation; 'deg', 'degFlat' and 'degFlatPos' are synonymous - as the conversion is done at the vertex level.

property degFlatPos

Values in units of 'degFlatPos'.

When dealing with positions/sizes in isolation; 'deg', 'degFlat' and 'degFlatPos' are synonymous - as the conversion is done at the vertex level.

property dimensions

How many dimensions (x, y, z) are specified?

property direction

Direction of vector (i.e. angle between vector and the horizontal plane).

property height

Value in units of 'height' (normalized to the height of the window).

property magnitude

Magnitude of vector (i.e. length of the line from vector to (0, 0) in pixels).

property monitor

The monitor used for calculations within this object (*~psychopy.monitors.Monitor*).

property norm

Value in units of 'norm' (normalized device coordinates).

property pix

Values in units of 'pix' (pixels).

property pt

Vector coordinates in 'pt' (points).

Points are commonly used in print media to define text sizes. One point is equivalent to 1/72 inches, or around 0.35 mm.

set (*value, units, win=None*)

validate (*value, units*)

Validate input values.

Ensures the values are in the correct format.

Returns Parameters *value* and *units*.

Return type *tuple*

class `psychopy.layout.Size` (*value, units, win=None*)

Class representing a size.

Parameters

- **value** (*ArrayLike*) – Array of values representing size axis-aligned bounding box within a coordinate system. Sizes are specified in a similar manner to *~psychopy.layout.Vector* as either 1xN for single vectors, and Nx2 or Nx3 for multiple positions.
- **units** (*str or None*) – Units which *value* has been specified in. Applicable values are 'pix', 'deg', 'degFlat', 'degFlatPos', 'cm', 'pt', 'norm', 'height', or *None*.
- **win** (*~psychopy.visual.Window or None*) – Window associated with this size object. This value must be specified if you wish to map sizes to coordinate systems that require additional information about the monitor the window is being displayed on.

property cm

Values in units of 'cm' (centimeters).

copy ()

Create a copy of this object

property deg

Values in units of 'deg' (degrees of visual angle).

property degFlat

Values in units of 'degFlat' (degrees of visual angle corrected for screen curvature).

When dealing with positions/sizes in isolation; 'deg', 'degFlat' and 'degFlatPos' are synonymous - as the conversion is done at the vertex level.

property degFlatPos

Values in units of 'degFlatPos'.

When dealing with positions/sizes in isolation; 'deg', 'degFlat' and 'degFlatPos' are synonymous - as the conversion is done at the vertex level.

property dimensions

How many dimensions (x, y, z) are specified?

property direction

Direction of vector (i.e. angle between vector and the horizontal plane).

property height

Value in units of 'height' (normalized to the height of the window).

property magnitude

Magnitude of vector (i.e. length of the line from vector to (0, 0) in pixels).

property monitor

The monitor used for calculations within this object (~*psychopy.monitors.Monitor*).

property norm

Value in units of 'norm' (normalized device coordinates).

property pix

Values in units of 'pix' (pixels).

property pt

Vector coordinates in 'pt' (points).

Points are commonly used in print media to define text sizes. One point is equivalent to 1/72 inches, or around 0.35 mm.

set (*value, units, win=None*)

validate (*value, units*)

Validate input values.

Ensures the values are in the correct format.

Returns Parameters *value* and *units*.

Return type tuple

class `psychopy.layout.Vertices` (*verts, obj=None, size=None, pos=None, units=None, flip=None, anchor=None*)

Class representing an array of vertices.

Parameters

- **verts** (*ArrayLike*) – Array of coordinates specifying the locations of vertices.

- **obj** (*object or None*) –
- **size** (*ArrayLike or None*) – Scaling factors for vertices along each dimension.
- **pos** (*ArrayLike or None*) – Offset for vertices along each dimension.
- **units** (*str or None*) – Units which *verts* has been specified in. Applicable values are ‘pix’, ‘deg’, ‘degFlat’, ‘degFlatPos’, ‘cm’, ‘pt’, ‘norm’, ‘height’, or *None*.
- **flip** (*ArrayLike or None*) – Array of boolean values specifying which dimensions to flip/mirror. Mirroring is applied prior to any other transformation.
- **anchor** (*str or None*) – Anchor location for vertices, specifies the origin for the vertices.

property anchor

Anchor location (*str*).

Possible values are on of ‘top’, ‘bottom’, ‘left’, ‘right’, ‘center’. Combinations of these values may also be specified (e.g., ‘top_center’, ‘center-right’, ‘topleft’, etc. are all valid).

property anchorAdjust

Map anchor values to numeric vertices adjustments.

property cm

Get absolute positions of vertices in ‘cm’ units.

property deg

Get absolute positions of vertices in ‘deg’ units.

property degFlat

Get absolute positions of vertices in ‘degFlat’ units.

property flip

1x2 array for flipping vertices along each axis; set as *True* to flip or *False* to not flip (*ArrayLike*).

If set as a single value, will duplicate across both axes. Accessing the protected attribute (*._flip*) will give an array of 1s and -1s with which to multiply vertices.

property flipHoriz

Apply horizontal mirroring (*bool*)?

property flipVert

Apply vertical mirroring (*bool*)?

getas (*units*)

property height

Get absolute positions of vertices in ‘height’ units.

property norm

Get absolute positions of vertices in ‘norm’ units.

property pix

Get absolute positions of vertices in ‘pix’ units.

property pos

Positional offset of the vertices (*~psychopy.layout.Vector* or *ArrayLike*).

setas (*value, units*)

property size

Scaling factors for vertices (*~psychopy.layout.Vector* or *ArrayLike*).

property units

Units which the vertices are specified in (*str*).

9.19 psychopy.logging - control what gets logged

Provides functions for logging error and other messages to one or more files and/or the console, using python's own logging module. Some warning messages and error messages are generated by PsychoPy itself. The user can generate more using the functions in this module.

There are various levels for logged messages with the following order of importance: ERROR, WARNING, DATA, EXP, INFO and DEBUG.

When setting the level for a particular log target (e.g. LogFile) the user can set the minimum level that is required for messages to enter the log. For example, setting a level of INFO will result in INFO, EXP, DATA, WARNING and ERROR messages to be recorded but not DEBUG messages.

By default, PsychoPy will record messages of WARNING level and above to the console. The user can silence that by setting it to receive only CRITICAL messages, (which PsychoPy doesn't use) using the commands:

```
from psychopy import logging
logging.console.setLevel(logging.CRITICAL)
```

class psychopy.logging.LogFile (*f=None, level=30, filemode='a', logger=None, encoding='utf8'*)

A text stream to receive inputs from the logging system

Create a log file as a target for logged entries of a given level

Parameters

- **f**: this could be a string to a path, that will be created if it doesn't exist. Alternatively this could be a file object, sys.stdout or any object that supports .write() and .flush() methods
- **level**: The minimum level of importance that a message must have to be logged by this target.
- **filemode**: 'a', 'w' Append or overwrite existing log file

setLevel (*level*)

Set a new minimal level for the log file/stream

write (*txt*)

Write directly to the log file (without using logging functions). Useful to send messages that only this file receives

class psychopy.logging._Logger (*format='% (t).4f \%(levelname)s \%(message)s'*)

Maintains a set of log targets (text streams such as files or stdout)

self.targets is a list of dicts { 'stream':stream, 'level':level }

The string-formatted elements *%(xxxx)f* can be used, where each *xxxx* is an attribute of the LogEntry. e.g. *t*, *t_ms*, *level*, *levelname*, *message*

addTarget (*target*)

Add a target, typically a LogFile to the logger

flush ()

Process all current messages to each target

log (*message, level, t=None, obj=None*)

Add the *message* to the log stack at the appropriate *level*

If no relevant targets (files or console) exist then the message is simply discarded.

removeTarget (*target*)

Remove a target, typically a `LogFile` from the logger

`psychopy.logging.addLevel` (*level, levelName*)

Associate 'levelName' with 'level'.

This is used when converting levels to text during message formatting.

`psychopy.logging.critical` (*message*)

Send the message to any receiver of logging info (e.g. a `LogFile`) of level `log.CRITICAL` or higher

`psychopy.logging.data` (*msg, t=None, obj=None*)

Log a message about data collection (e.g. a key press)

usage:: `log.data(message)`

Sends the message to any receiver of logging info (e.g. a `LogFile`) of level `log.DATA` or higher

`psychopy.logging.debug` (*msg, t=None, obj=None*)

Log a debugging message (not likely to be wanted once experiment is finalised)

usage:: `log.debug(message)`

Sends the message to any receiver of logging info (e.g. a `LogFile`) of level `log.DEBUG` or higher

`psychopy.logging.error` (*message*)

Send the message to any receiver of logging info (e.g. a `LogFile`) of level `log.ERROR` or higher

`psychopy.logging.exp` (*msg, t=None, obj=None*)

Log a message about the experiment (e.g. a new trial, or end of a stimulus)

usage:: `log.exp(message)`

Sends the message to any receiver of logging info (e.g. a `LogFile`) of level `log.EXP` or higher

`psychopy.logging.fatal` (*msg, t=None, obj=None*)

`log.critical(message)` Send the message to any receiver of logging info (e.g. a `LogFile`) of level `log.CRITICAL` or higher

`psychopy.logging.flush` (*logger=<psychopy.logging._Logger object>*)

Send current messages in the log to all targets

`psychopy.logging.getLevel` (*level*)

Return the textual representation of logging level 'level'.

If the level is one of the predefined levels (`CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG`) then you get the corresponding string. If you have associated levels with names using `addLevelName` then the name you have associated with 'level' is returned.

If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned.

Otherwise, the string "Level %s" % level is returned.

`psychopy.logging.info` (*msg, t=None, obj=None*)

Log some information - maybe useful, maybe not

usage:: `log.info(message)`

Sends the message to any receiver of logging info (e.g. a `LogFile`) of level `log.INFO` or higher

`psychopy.logging.log` (*msg, level, t=None, obj=None*)

Log a message

usage:: log(msg, level, t=t, obj=obj)

Log the msg, at a given level on the root logger

psychoPy.logging.**setDefaultClock** (*clock*)

Set the default clock to be used to reference all logging times. Must be a *psychoPy.core.Clock* object. Beware that if you reset the clock during the experiment then the resets will be reflected here. That might be useful if you want your logs to be reset on each trial, but probably not.

psychoPy.logging.**warn** (*msg, t=None, obj=None*)

log.warning(message)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.WARNING* or higher

psychoPy.logging.**warning** (*message*)

Sends the message to any receiver of logging info (e.g. a LogFile) of level *log.WARNING* or higher

9.19.1 flush()

psychoPy.logging.**flush** (*logger=<psychoPy.logging._Logger object>*)

Send current messages in the log to all targets

9.19.2 setDefaultClock()

psychoPy.logging.**setDefaultClock** (*clock*)

Set the default clock to be used to reference all logging times. Must be a *psychoPy.core.Clock* object. Beware that if you reset the clock during the experiment then the resets will be reflected here. That might be useful if you want your logs to be reset on each trial, but probably not.

9.20 psychoPy.microphone - Capture and analyze sound

(Available as of version 1.74.00; Advanced features available as of 1.77.00)

Deprecated Use *Microphone* for new projects.

9.20.1 Overview

AudioCapture() allows easy audio recording and saving of arbitrary sounds to a file (wav format). AudioCapture will likely be replaced entirely by AdvAudioCapture in the near future.

AdvAudioCapture() can do everything AudioCapture does, and also allows onset-marker sound insertion and detection, loudness computation (RMS audio “power”), and lossless file compression (flac). The Builder microphone component now uses AdvAudioCapture by default.

9.20.2 Audio Capture

`psychopy.microphone.switchOn` (*sampleRate=48000, outputDevice=None, bufferSize=None*)

You need to switch on the microphone before use, which can take several seconds. The only time you can specify the sample rate (in Hz) is during `switchOn()`.

Considerations on the default sample rate 48kHz:

```
DVD or video = 48,000
CD-quality   = 44,100 / 24 bit
human hearing: ~15,000 (adult); children & young adult higher
human speech: 100-8,000 (useful for telephone: 100-3,300)
Google speech API: 16,000 or 8,000 only
Nyquist frequency: twice the highest rate, good to oversample a bit
```

pyo's `downsamp()` function can reduce 48,000 to 16,000 in about 0.02s (uses integer steps sizes). So recording at 48kHz will generate high-quality archival data, and permit easy downsampling.

outputDevice, bufferSize: set these parameters on the `pyoSndServer` before booting; None means use pyo's default values

class `psychopy.microphone.AdvAudioCapture` (*name='advMic', filename="", saveDir="", sampletype=0, buffering=16, chnl=0, stereo=True, autoLog=True*)

Class extends `AudioCapture`, plays marker sound as a “start” indicator.

Has method for retrieving the marker onset time from the file, to allow calculation of vocal RT (or other sound-based RT).

See Coder demo > input > `latencyFromTone.py`

Parameters

name : Stem for the output file, also used in logging.

filename : optional file name to use; default = ‘name-onsetTimeEpoch.wav’

saveDir : Directory to use for output .wav files. If a `saveDir` is given, it will return ‘saveDir/file’. If no `saveDir`, then return `abspath(file)`

sampletype [bit depth] pyo recording option: 0=16 bits int, 1=24 bits int; 2=32 bits int

buffering [pyo argument] Controls the buffering argument for pyo if necessary

chnl [int (default=0)] which audio input channel to record (default=0)

stereo [bool or nChannels (default = True)] how many channels to record

compress (*keep=False*)

Compress using FLAC (lossless compression).

getLoudness ()

Return the RMS loudness of the saved recording.

getMarkerInfo ()

Returns (hz, duration, volume) of the marker sound. Custom markers always return 0 hz (regardless of the sound).

getMarkerOnset (*chunk=128, secs=0.5, filename=""*)

Return (onset, offset) time of the first marker within the first *secs* of the saved recording.

Has approx ~1.33ms resolution at 48000Hz, `chunk=64`. Larger chunks can speed up processing times, at a sacrifice of some resolution, e.g., to pre-process long recordings with multiple markers.

If given a filename, it will first set that file as the one to work with, and then try to detect the onset marker.

playMarker ()

Plays the current marker sound. This is automatically called at the start of recording, but can be called anytime to insert a marker.

playback (*block=True, loops=0, stop=False, log=True*)

Plays the saved .wav file, as just recorded or resampled. Execution blocks by default, but can return immediately with *block=False*.

loops : number of extra repetitions; 0 = play once

stop : True = immediately stop ongoing playback (if there is one), and return

record (*sec, filename="", block=False*)

Starts recording and plays an onset marker tone just prior to returning. The idea is that the start of the tone in the recording indicates when this method returned, to enable you to sync a known recording onset with other events.

resample (*newRate=16000, keep=True, log=True*)

Re-sample the saved file to a new rate, return the full path.

Can take several visual frames to resample a 2s recording.

The default values for `resample()` are for Google-speech, keeping the original (presumably recorded at 48kHz) to archive. A warning is generated if the new rate not an integer factor / multiple of the old rate.

To control anti-aliasing, use `pyo.downsamp()` or `upsamp()` directly.

reset (*log=True*)

Restores to fresh state, ready to record again

setFile (*filename*)

Sets the name of the file to work with.

setMarker (*tone=19000, secs=0.015, volume=0.03, log=True*)

Sets the onset marker, where *tone* is either in hz or a custom sound.

The default tone (19000 Hz) is recommended for auto-detection, as being easier to isolate from speech sounds (and so reliable to detect). The default duration and volume are appropriate for a quiet setting such as a lab testing room. A louder volume, longer duration, or both may give better results when recording loud sounds or in noisy environments, and will be auto-detected just fine (even more easily). If the hardware microphone in use is not physically near the speaker hardware, a louder volume is likely to be required.

Custom sounds cannot be auto-detected, but are supported anyway for presentation purposes. E.g., a recording of someone saying “go” or “stop” could be passed as the onset marker.

stop (*log=True*)

Interrupt a recording that is in progress; close & keep the file.

Ends the recording before the duration that was initially specified. The same file name is retained, with the same onset time but a shorter duration.

The same recording cannot be resumed after a stop (it is not a pause), but you can start a new one.

uncompress (*keep=False*)

Uncompress from FLAC to .wav format.

9.20.3 Speech recognition

Google's speech to text API is no longer available. AT&T, IBM, and Wit.ai have a similar (paid) service.

9.20.4 Misc

Functions for file-oriented Discrete Fourier Transform and RMS computation are also provided.

`psychopy.microphone.wav2flac` (*path*, *keep=True*, *level=5*)

Lossless compression: convert .wav file (on disk) to .flac format.

If *path* is a directory name, convert all .wav files in the directory.

keep to retain the original .wav file(s), default *True*.

level is compression level: 0 is fastest but larger, 8 is slightly smaller but much slower.

`psychopy.microphone.flac2wav` (*path*, *keep=True*)

Uncompress: convert .flac file (on disk) to .wav format (new file).

If *path* is a directory name, convert all .flac files in the directory.

keep to retain the original .flac file(s), default *True*.

`psychopy.microphone.getDft` (*data*, *sampleRate=None*, *wantPhase=False*)

Compute and return magnitudes of `numpy.fft.fft()` of the data.

If given a sample rate (samples/sec), will return (magn, freq). If *wantPhase* is *True*, phase in radians is also returned (magn, freq, phase). data should have power-of-2 samples, or will be truncated.

`psychopy.microphone.getRMS` (*data*)

Compute and return the audio power ("loudness").

Uses `numpy.std()` as RMS. `std()` is same as RMS if the mean is 0, and .wav data should have a mean of 0. Returns an array if given stereo data (RMS computed within-channel).

data can be an array (1D, 2D) or filename; .wav format only. data from .wav files will be normalized to -1..+1 before RMS is computed.

9.21 psychopy.misc - miscellaneous routines for converting units etc

Wrapper for all miscellaneous functions and classes from `psychopy.tools`

`psychopy.misc` has gradually grown very large and the underlying code for its functions are distributed in multiple files. You can still (at least for now) import the functions here using `from psychopy import misc` but you can also import them from the `tools` sub-modules.

9.21.1 From `psychopy.tools.filetools`

<code>toFile(filename, data)</code>	Save data (of any sort) as a pickle file.
<code>fromFile(filename[, encoding])</code>	Load data from a psydat, pickle or JSON file.
<code>mergeFolder(src, dst[, pattern])</code>	Merge a folder into another.

9.21.2 From `psychopy.tools.colorspectools`

<code>dkl2rgb(dkl[, conversionMatrix])</code>	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.
<code>dklCart2rgb(LUM, LM, S[, conversionMatrix])</code>	Like <code>dkl2rgb</code> except that it uses cartesian coords (LM,S,LUM) rather than spherical coords for DKL (elev, azim, contr).
<code>rgb2dklCart(picture[, conversionMatrix])</code>	Convert an RGB image into Cartesian DKL space.
<code>hsv2rgb(hsv_Nx3)</code>	Convert from HSV color space to RGB gun values.
<code>lms2rgb(lms_Nx3[, conversionMatrix])</code>	Convert from cone space (Long, Medium, Short) to RGB.
<code>rgb2lms(rgb_Nx3[, conversionMatrix])</code>	Convert from RGB to cone space (LMS).
<code>dkl2rgb(dkl[, conversionMatrix])</code>	Convert from DKL color space (Derrington, Krauskopf & Lennie) to RGB.

9.21.3 From `psychopy.tools.coordinatetools`

<code>cart2pol(x, y[, units])</code>	Convert from cartesian to polar coordinates.
<code>cart2sph(z, y, x)</code>	Convert from cartesian coordinates (x,y,z) to spherical (elevation, azimuth, radius).
<code>pol2cart(theta, radius[, units])</code>	Convert from polar to cartesian coordinates.
<code>sph2cart(*args)</code>	Convert from spherical coordinates (elevation, azimuth, radius) to cartesian (x,y,z).

9.21.4 From `psychopy.tools.monitorunittools`

<code>convertToPix(vertices, pos, units, win)</code>	Takes vertices and position, combines and converts to pixels from any unit
<code>cm2pix(cm, monitor)</code>	Convert size in cm to size in pixels for a given Monitor object.
<code>cm2deg(cm, monitor[, correctFlat])</code>	Convert size in cm to size in degrees for a given Monitor object
<code>deg2cm(degrees, monitor[, correctFlat])</code>	Convert size in degrees to size in pixels for a given Monitor object.
<code>deg2pix(degrees, monitor[, correctFlat])</code>	Convert size in degrees to size in pixels for a given Monitor object
<code>pix2cm(pixels, monitor)</code>	Convert size in pixels to size in cm for a given Monitor object
<code>pix2deg(pixels, monitor[, correctFlat])</code>	Convert size in pixels to size in degrees for a given Monitor object

9.21.5 From `psychopy.tools.imagetools`

<code>array2image(a)</code>	Takes an array and returns an image object (PIL).
<code>image2array(im)</code>	Takes an image object (PIL) and returns a numpy array.
<code>makeImageAuto(inarray)</code>	Combines float_uint8 and image2array operations ie.

9.21.6 From `psychopy.tools.plottools`

<code>plotFrameIntervals(intervals)</code>	Plot a histogram of the frame intervals.
--	--

9.21.7 From `psychopy.tools.typetools`

<code>float_uint8(inarray)</code>	Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint8s ranging 0:255
<code>uint8_float(inarray)</code>	Converts arrays, lists, tuples and UINTs ranging 0:255 into an array of floats ranging -1:1
<code>float_uint16(inarray)</code>	Converts arrays, lists, tuples and floats ranging -1:1 into an array of Uint16s ranging 0:2^16

9.21.8 From `psychopy.tools.unittools`

<code>radians</code>	<code>radians(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])</code>
<code>degrees</code>	<code>degrees(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])</code>

9.22 `psychopy.monitors` - for those that don't like Monitor Center

Most users won't need to use the code here. In general the Monitor Centre interface is sufficient and monitors setup that way can be passed as strings to `Window`s. If there is some aspect of the normal calibration that you wish to override. eg:

```
from psychopy import visual, monitors
mon = monitors.Monitor('SonyG55') #fetch the most recent calib for this monitor
mon.setDistance(114) #further away than normal?
win = visual.Window(size=[1024,768], monitor=mon)
```

You might also want to fetch the `Photometer` class for conducting your own calibrations

9.22.1 Monitor

```
class psychopy.monitors.Monitor (name, width=None, distance=None, gamma=None,
                                   notes=None, useBits=None, verbose=True, current-
                                   Calib=None, autoLog=True)
```

Creates a monitor object for storing calibration details. This will be loaded automatically from disk if the monitor name is already defined (see methods).

Many settings from the stored monitor can easily be overridden either by adding them as arguments during the initial call.

arguments:

- *width*, *distance*, *gamma* are details about the calibration
- *notes* is a text field to store any useful info
- *useBits* True, False, None
- *verbose* True, False, None
- **currentCalib** is a dictionary object containing various fields for a calibration. Use with caution since the dictionary may not contain all the necessary fields that a monitor object expects to find.

eg:

```
myMon = Monitor('sony500', distance=114) Fetches the info on the sony500 and overrides its
usual distance to be 114cm for this experiment.
```

These can be saved to the monitor file using *save()* or not (in which case the changes will be lost)

loadAll()

Fetches the calibrations for this monitor from disk, storing them as *self.calibs*

copyCalib (calibName=None)

Stores the settings for the current calibration settings as new monitor.

delCalib (calibName)

Remove a specific calibration from the current monitor. Won't be finalised unless monitor is saved

gammaIsDefault()

Determine whether we're using the default gamma values

getCalibDate()

As a python date object (convert to string using *calibTools.strFromDate*)

getDKL_RGB (RECOMPUTE=False)

Returns the DKL->RGB conversion matrix. If one has been saved this will be returned. Otherwise, if power spectra are available for the monitor a matrix will be calculated.

getDistance()

Returns distance from viewer to the screen in cm, or None if not known

getGamma()

Returns just the gamma value (not the whole grid)

getGammaGrid()

Gets the min,max,gamma values for the each gun

getLMS_RGB (recompute=False)

Returns the LMS->RGB conversion matrix. If one has been saved this will be returned. Otherwise (if power spectra are available for the monitor) a matrix will be calculated.

getLevelsPost()

Gets the measured luminance values from last calibration TEST

getLevelsPre ()
 Gets the measured luminance values from last calibration

getLinearizeMethod ()
 Gets the method that this monitor is using to linearize the guns

getLumsPost ()
 Gets the measured luminance values from last calibration TEST

getLumsPre ()
 Gets the measured luminance values from last calibration

getMeanLum ()
 Returns the mean luminance of the screen if explicitly stored

getNotes ()
 Notes about the calibration

getPsychopyVersion ()
 Returns the version of PsychoPy that was used to create this calibration

getSizePix ()
 Returns the size of the current calibration in pixels, or None if not defined

getSpectra ()
 Gets the wavelength values from the last spectrometer measurement (if available)

usage:

- nm, power = monitor.getSpectra()

getUseBits ()
 Was this calibration carried out with a bits++ box

getWidth ()
 Of the viewable screen in cm, or None if not known

lineariseLums (*desiredLums*, *newInterpolators=False*, *overrideGamma=None*)
 Equivalent of *linearizeLums* ().

linearizeLums (*desiredLums*, *newInterpolators=False*, *overrideGamma=None*)
 lums should be uncalibrated luminance values (e.g. a linear ramp) ranging 0:1

newCalib (*calibName=None*, *width=None*, *distance=None*, *gamma=None*, *notes=None*,
useBits=False, *verbose=True*)
 create a new (empty) calibration for this monitor and makes this the current calibration

save ()
 Save the current calibrations to disk.

This will write a *json* file to the *monitors* subfolder of your PsychoPy configuration folder (typically *~/psychopy3/monitors* on Linux and macOS, and *%APPDATA%\psychopy3monitors* on Windows).

saveMon ()
 Equivalent of *save* ().

setCalibDate (*date=None*)
 Sets the current calibration to have a date/time or to the current date/time if none given. (Also returns the date as set)

setCurrent (*calibration=-1*)
 Sets the current calibration for this monitor. Note that a single file can hold multiple calibrations each stored under a different key (the date it was taken)

The argument is either a string (naming the calib) or an integer eg:

`myMon.setCurrent('mainCalib')` fetches the calibration named `mainCalib`. You can name calibrations what you want but PsychoPy will give them names of date/time by default. In Monitor Center you can 'copy...' a calibration and give it a new name to keep a second version.

`calibName = myMon.setCurrent(0)` fetches the first calibration (alphabetically) for this monitor

`calibName = myMon.setCurrent(-1)` fetches the last **alphabetical** calibration for this monitor (this is default). If default names are used for calibrations (ie date/time stamp) then this will import the most recent.

setDKL_RGB (*dkl_rgb*)

Sets the DKL->RGB conversion matrix for a chromatically calibrated monitor (matrix is a 3x3 num array).

setDistance (*distance*)

To the screen (cm)

setGamma (*gamma*)

Sets the gamma value(s) for the monitor. This only uses a single gamma value for the three guns, which is fairly approximate. Better to use `setGammaGrid` (which uses one gamma value for each gun)

setGammaGrid (*gammaGrid*)

Sets the min,max,gamma values for the each gun

setLMS_RGB (*lms_rgb*)

Sets the LMS->RGB conversion matrix for a chromatically calibrated monitor (matrix is a 3x3 num array).

setLevelsPost (*levels*)

Sets the last set of luminance values measured AFTER calibration

setLevelsPre (*levels*)

Sets the last set of luminance values measured during calibration

setLineariseMethod (*method*)

Sets the method for linearising 0 uses $y=a+(bx)^{\text{gamma}}$ 1 uses $y=(a+bx)^{\text{gamma}}$ 2 uses linear interpolation over the curve

setLumsPost (*lums*)

Sets the last set of luminance values measured AFTER calibration

setLumsPre (*lums*)

Sets the last set of luminance values measured during calibration

setMeanLum (*meanLum*)

Records the mean luminance (for reference only)

setNotes (*notes*)

For you to store notes about the calibration

setPsychopyVersion (*version*)

To store the version of PsychoPy that this calibration used

setSizePix (*pixels*)

Set the size of the screen in pixels x,y

setSpectra (*nm, rgb*)

Sets the phosphor spectra measured by the spectrometer

setUseBits (*usebits*)

DEPRECATED: Use the new hardware classes to control these devices

setWidth (*width*)

Of the viewable screen (cm)

9.22.2 GammaCalculator

class psychopy.monitors.**GammaCalculator** (*inputs=(), lums=(), gamma=None, bitsIN=8, bitsOUT=8, eq=1*)

Class for managing gamma tables

Parameters:

- **inputs (required)= values at which you measured screen luminance either** in range 0.0:1.0, or range 0:255. Should include the min and max of the monitor

Then give EITHER “lums” or “gamma”:

- lums = measured luminance at given input levels
- gamma = your own gamma value (single float)
- bitsIN = number of values in your lookup table
- bitsOUT = number of bits in the DACs

myTable.gammaModel myTable.gamma

fitGammaErrFun (*params, x, y, minLum, maxLum*)

Provides an error function for fitting gamma function

(used by fitGammaFun)

fitGammaFun (*x, y*)

Fits a gamma function to the monitor calibration data.

Parameters: -xVals are the monitor look-up-table vals, either 0-255 or 0.0-1.0 -yVals are the measured luminances from a photometer/spectrometer

9.22.3 getAllMonitors ()

psychopy.monitors.**getAllMonitors** ()

Find the names of all monitors for which calibration files exist

9.22.4 getLumSeriesPR650 ()

psychopy.monitors.**getLumSeriesPR650** (*lumLevels=8, winSize=800, 600, monitor=None, gamma=1.0, allGuns=True, useBits=False, autoMode='auto', stimSize=0.3, photometer='COM1'*)

DEPRECATED (since v1.60.01): Use psychopy.monitors.getLumSeries () instead

9.22.5 `getRGBspectra()`

`psychoPy.monitors.getRGBspectra(stimSize=0.3, winSize=800, 600, photometer='COM1')`

usage: `getRGBspectra(stimSize=0.3, winSize=(800,600), photometer='COM1')`

Params

- 'photometer' could be a photometer object or a serial port name on which a photometer might be found (not recommended)

9.22.6 `gammaFun()`

`psychoPy.monitors.gammaFun(xx, minLum, maxLum, gamma, eq=1, a=None, b=None, k=None)`

Returns gamma-transformed luminance values. $y = \text{gammaFun}(x, \text{minLum}, \text{maxLum}, \text{gamma})$

a and b are calculated directly from minLum, maxLum, gamma

Parameters:

- **xx** are the input values (range 0-255 or 0.0-1.0)
- **minLum** = the minimum luminance of your monitor
- **maxLum** = the maximum luminance of your monitor (for this gun)
- **gamma** = the value of gamma (for this gun)

9.22.7 `gammaInvFun()`

`psychoPy.monitors.gammaInvFun(yy, minLum, maxLum, gamma, b=None, eq=1)`

Returns inverse gamma function for desired luminance values. $x = \text{gammaInvFun}(y, \text{minLum}, \text{maxLum}, \text{gamma})$

a and b are calculated directly from minLum, maxLum, gamma **Parameters:**

- **xx** are the input values (range 0-255 or 0.0-1.0)
- **minLum** = the minimum luminance of your monitor
- **maxLum** = the maximum luminance of your monitor (for this gun)
- **gamma** = the value of gamma (for this gun)
- **eq determines the gamma equation used;** `eq==1`[default]: $yy = a + (b * xx)**\text{gamma}$ `eq==2`: $yy = (a + b*xx)**\text{gamma}$

9.22.8 `makeDKL2RGB()`

`psychoPy.monitors.makeDKL2RGB(nm, powerRGB)`

Creates a 3x3 DKL->RGB conversion matrix from the spectral input powers

9.22.9 makeLMS2RGB ()

`psychoPy.monitors.makeLMS2RGB (nm, powerRGB)`
 Creates a 3x3 LMS->RGB conversion matrix from the spectral input powers

9.23 psychoPy.parallel - functions for interacting with the parallel port

This module provides read / write access to the parallel port for Linux or Windows.

The `Parallel` class described below will attempt to load whichever parallel port driver is first found on your system and should suffice in most instances. If you need to use a specific driver then, instead of using `ParallelPort` shown below you can use one of the following as drop-in replacements, forcing the use of a specific driver:

- `psychoPy.parallel.PParallelInpOut`
- `psychoPy.parallel.PParallelDLPortIO`
- `psychoPy.parallel.PParallelLinux`

Either way, each instance of the class can provide access to a different parallel port.

There is also a legacy API which consists of the routines which are directly in this module. That API assumes you only ever want to use a single parallel port at once.

`psychoPy.parallel.ParallelPort`
 alias of `psychoPy.parallel._linux.PParallelLinux`

9.23.1 Legacy functions

We would strongly recommend you use the class above instead: these are provided for backwards compatibility only.

`parallel.setPortAddress ()`
 Set the memory address or device node for your parallel port of your parallel port, to be used in subsequent commands

Common port addresses:

```
LPT1 = 0x0378 or 0x03BC
LPT2 = 0x0278 or 0x0378
LPT3 = 0x0278
```

or for Linux:: /dev/parport0

This routine will attempt to find a usable driver depending on your platform

`parallel.setData ()`
 Set the data to be presented on the parallel port (one ubyte). Alternatively you can set the value of each pin (data pins are pins 2-9 inclusive) using `setPin ()`

Examples:

```
parallel.setData(0) # sets all pins low
parallel.setData(255) # sets all pins high
parallel.setData(2) # sets just pin 3 high (remember that pin2=bit0)
parallel.setData(3) # sets just pins 2 and 3 high
```

You can also convert base 2 to int v easily in python:

```
parallel.setData(int("00000011", 2)) # pins 2 and 3 high
parallel.setData(int("00000101", 2)) # pins 2 and 4 high
```

`parallel.setPin(state)`

Set a desired pin to be high (1) or low (0).

Only pins 2-9 (incl) are normally used for data output:

```
parallel.setPin(3, 1) # sets pin 3 high
parallel.setPin(3, 0) # sets pin 3 low
```

`parallel.readPin()`

Determine whether a desired (input) pin is high(1) or low(0).

Pins 2-13 and 15 are currently read here

9.24 psychopy.plugins - utilities for extending with plugins

9.24.1 Overview

<code>loadPlugin(plugin, *args, **kwargs)</code>	Load a plugin to extend PsychoPy.
<code>listPlugins([which])</code>	Get a list of installed or loaded PsychoPy plugins.
<code>startUpPlugins(plugins[, add, verify])</code>	Specify which plugins should be loaded automatically when a PsychoPy session starts.
<code>computeChecksum(fpath[, method, writeOut])</code>	Compute the checksum hash/key for a given package.

9.24.2 Details

`psychopy.plugins.loadPlugin(plugin, *args, **kwargs)`

Load a plugin to extend PsychoPy.

Plugins are packages which extend upon PsychoPy's existing functionality by dynamically importing code at runtime, without modifying the existing installation files. Plugins create or redefine objects in the namespaces of modules (eg. *psychopy.visual*) and unbound classes, allowing them to be used as if they were part of PsychoPy. In some cases, objects exported by plugins will be registered for a particular function if they define entry points into specific modules.

Plugins are simply Python packages, `loadPlugin` will search for them in directories specified in `sys.path`. Only packages which define entry points in their metadata which pertain to PsychoPy can be loaded with this function. This function also permits passing optional arguments to a callable object in the plugin module to run any initialization routines prior to loading entry points.

This function is robust, simply returning `True` or `False` whether a plugin has been fully loaded or not. If a plugin fails to load, the reason for it will be written to the log as a warning or error, and the application will continue running. This may be undesirable in some cases, since features the plugin provides may be needed at some point and would lead to undefined behavior if not present. If you want to halt the application if a plugin fails to load, consider using `requirePlugin()`.

It is advised that you use this function only when using PsychoPy as a library. If using the builder or coder GUI, it is recommended that you use the plugin dialog to enable plugins for PsychoPy sessions spawned by the experiment runner. However, you can still use this function if you want to load additional plugins for a given

experiment, having their effects isolated from the main application and other experiments.

Parameters

- **plugin** (*str*) – Name of the plugin package to load. This usually refers to the package or project name.
- ***args** – Optional arguments and keyword arguments to pass to the plugin’s `__register__` function.
- ****kwargs** – Optional arguments and keyword arguments to pass to the plugin’s `__register__` function.

Returns *True* if the plugin has valid entry points and was loaded successfully. Also returns *True* if the plugin was already loaded by a previous `loadPlugin` call this session, this function will have no effect in this case. *False* is returned if the plugin defines no entry points specific to PsychoPy or crashed (an error is logged).

Return type `bool`

Warning: Make sure that plugins installed on your system are from reputable sources, as they may contain malware! PsychoPy is not responsible for undefined behaviour or bugs associated with the use of 3rd party plugins.

See also:

`listPlugins()` Search for and list installed or loaded plugins.

`requirePlugin()` Require a plugin be previously loaded.

Examples

Load a plugin by specifying its package/project name:

```
loadPlugin('psychopy-hardware-box')
```

You can give arguments to this function which are passed on to the plugin:

```
loadPlugin('psychopy-hardware-box', switchOn=True, baudrate=9600)
```

You can use the value returned from `loadPlugin` to determine if the plugin is installed and supported by the platform:

```
hasPlugin = loadPlugin('psychopy-hardware-box')
if hasPlugin:
    # initialize objects which require the plugin here ...
```

`psychopy.plugins.listPlugins` (*which='all'*)

Get a list of installed or loaded PsychoPy plugins.

This function lists either all potential plugin packages installed on the system, those registered to be loaded automatically when PsychoPy starts, or those that have been previously loaded successfully this session.

Parameters **which** (*str*) – Category to list plugins. If ‘all’, all plugins installed on the system will be listed, whether they have been loaded or not. If ‘loaded’, only plugins that have been previously loaded successfully this session will be listed. If ‘startup’, plugins registered to be loaded when a PsychoPy session starts will be listed, whether or not they have been loaded this

session. If 'unloaded', plugins that have not been loaded but are installed will be listed. If 'failed', returns a list of plugin names that attempted to load this session but failed for some reason.

Returns Names of PsychoPy related plugins as strings. You can load all installed plugins by passing list elements to *loadPlugin*.

Return type *list*

See also:

loadPlugin() Load a plugin into the current session.

Examples

Load all plugins installed on the system into the current session (assumes all plugins don't require any additional arguments passed to them):

```
for plugin in plugins.listPlugins():
    plugins.loadPlugin(plugin)
```

If certain plugins take arguments, you can do this give specific arguments when loading all plugins:

```
pluginArgs = {'some-plugin': (('someArg',), {'setup': True, 'spam': 10})}
for plugin in plugins.listPlugins():
    try:
        args, kwargs = pluginArgs[plugin]
        plugins.loadPlugin(plugin, *args, **kwargs)
    except KeyError:
        plugins.loadPlugin(plugin)
```

Check if a plugin package named *plugin-test* is installed on the system and has entry points into PsychoPy:

```
if 'plugin-test' in plugins.listPlugins():
    print("Plugin installed!")
```

Check if all plugins registered to be loaded on startup are currently active:

```
if not all([p in listPlugins('loaded') for p in listPlugins('startup')]):
    print('Please restart your PsychoPy session for plugins to take effect.')
```

`psychopy.plugins.startUpPlugins (plugins, add=True, verify=True)`

Specify which plugins should be loaded automatically when a PsychoPy session starts.

This function edits `psychopy.preferences.prefs.general['startUpPlugins']` and provides a means to verify if entries are valid. The PsychoPy session must be restarted for the plugins specified to take effect.

If using PsychoPy as a library, this function serves as a convenience to avoid needing to explicitly call *loadPlugin()* every time to use your favorite plugins.

Parameters

- **plugins** (*str, list or None*) – Name(s) of plugins to have load on startup.
- **add** (*bool*) – If *True* names of plugins will be appended to *startUpPlugins* unless a name is already present. If *False*, *startUpPlugins* will be set to *plugins*, overwriting the previous value. If *add=False* and *plugins=[]* or *plugins=None*, no plugins will be loaded in the next session.

- **verify** (*bool*) – Check if *plugins* are installed and have valid entry points to PsychoPy. Raises an error if any are not. This prevents undefined behavior arising from invalid plugins being loaded in the next session. If *False*, plugin names will be added regardless if they are installed or not.

Raises `RuntimeError` – If *verify=True*, any of *plugins* is not installed or does not have entry points to PsychoPy. This is raised to prevent issues in future sessions where invalid plugins are written to the config file and are automatically loaded.

Warning: Do not use this function within the builder or coder GUI! Use the plugin dialog to specify which plugins to load on startup. Only use this function when using PsychoPy as a library!

Examples

Adding plugins to load on startup:

```
startUpPlugins(['plugin1', 'plugin2'])
```

Clearing the startup plugins list, no plugins will be loaded automatically at the start of the next session:

```
plugins.startUpPlugins([], add=False)
# or ..
plugins.startUpPlugins(None, add=False)
```

If passing *None* or an empty list with *add=True*, the present value of *prefs.general['startUpPlugins']* will remain as-is.

`psychopy.plugins.computeChecksum` (*fpath*, *method='sha256'*, *writeOut=None*)

Compute the checksum hash/key for a given package.

Authors of PsychoPy plugins can use this function to compute a checksum hash and users can use it to check the integrity of their packages.

Parameters

- **fpath** (*str*) – Path to the plugin package or file.
- **method** (*str*) – Hashing method to use, values are ‘md5’ or ‘sha256’. Default is ‘sha256’.
- **writeOut** (*str*) – Path to a text file to write checksum data to. If the file exists, the data will be written as a line at the end of the file.

Returns Checksum hash digested to hexadecimal format.

Return type *str*

Examples

Compute a checksum for a package and write it to a file:

```
with open('checksum.txt', 'w') as f:
    f.write(computeChecksum(
        '/path/to/plugin/psychopy_plugin-1.0-py3.6.egg'))
```

9.25 psychopy.preferences - getting and setting preferences

You can set preferences on a per-experiment basis. For example, if you would like to use a specific audio library, but don't want to touch your user settings in general, you can import preferences and set the option *audioLib* accordingly:

```
from psychopy import prefs
prefs.hardware['audioLib'] = ['pyo']
from psychopy import sound
```

!!IMPORTANT!! You must import the sound module **AFTER** setting the preferences. To check that you are getting what you want (don't do this in your actual experiment):

```
print sound.Sound
```

The output should be `<class 'psychopy.sound.SoundPyo'>` for pyo, or `<class 'psychopy.sound.SoundPygame'>` for pygame.

You can find the names of the *preferences sections and their different options here*.

9.25.1 Preferences

Class for loading / saving prefs

class psychopy.preferences.Preferences

Users can alter preferences from the dialog box in the application, by editing their user preferences file (which is what the dialog box does) or, within a script, preferences can be controlled like this:

```
import psychopy
psychopy.prefs.hardware['audioLib'] = ['PTB', 'pyo', 'pygame']
print(prefs)
# prints the location of the user prefs file and all the current vals
```

Use the instance of *prefs*, as above, rather than the *Preferences* class directly if you want to affect the script that's running.

loadAll ()

Load the user prefs and the application data

loadUserPrefs ()

load user prefs, if any; don't save to a file because doing so will break *easy_install*. Saving to files within the *psychopy/* is fine, eg for key-bindings, but outside it (where user prefs will live) is not allowed by *easy_install* (security risk)

resetPrefs ()

removes *userPrefs.cfg*, does not touch *appData.cfg*

restoreBadPrefs (*cfg, result*)

result = result of validate

saveAppData ()

Save the various setting to the appropriate files (or discard, in some cases)

saveUserPrefs ()

Validate and save the various setting to the appropriate files (or discard, in some cases)

validate ()

Validate (user) preferences and reset invalid settings to defaults

9.26 psychopy.serial - functions for interacting with the serial port

is compatible with Chris Liechti's `pyserial` package. You can use it like this:

```
import serial
ser = serial.Serial(0, 19200, timeout=1) # open first serial port
#ser = serial.Serial('/dev/ttyS1', 19200, timeout=1)#or something like this for Mac/
↳Linux machines
ser.write('someCommand')
line = ser.readline() # read a '\n' terminated line
ser.close()
```

Ports are fully configurable with all the options you would expect of RS232 communications. See <https://pyserial.readthedocs.io> for further details and documentation.

`pyserial` is packaged in the Standalone (Windows and Mac distributions), for manual installations you should install this yourself.

9.27 psychopy.voicekey - Real-time sound processing

(Available as of version 1.83.00)

9.27.1 Overview

Hardware voice-keys are used to detect and signal acoustic properties in real time, e.g., the onset of a spoken word in word-naming studies. provides two virtual voice-keys, one for detecting vocal onsets and one for vocal offsets.

All voice-keys can take their input from a file or from a microphone. Event detection is typically quite similar in both cases.

The base class is very general, and is best thought of as providing a toolkit for developing a wide range of custom voice-keys. It would be possible to develop a set of voice-keys, each optimized for detecting different initial phonemes. Band-pass filtered data and zero-crossing counts are computed in real-time every 2ms.

9.27.2 Voice-Keys

class `psychopy.voicekey.OnsetVoiceKey` (*sec=0, file_out="", file_in="", **config*)

Class for speech onset detection.

Uses bandpass-filtered signal (100-3000Hz). When the voice key trips, the best voice-onset RT estimate is saved as *self.event_onset*, in sec.

Parameters

sec: duration to record in seconds

file_out: name for output filename (for microphone input)

file_in: name of input file for sound source (not microphone)

config: kwargs dict of parameters for configuration. defaults are:

 'msPerChunk': 2; duration of each real-time analysis chunk, in ms

 'signaler': default None

'autosave': True; False means manual saving to a file is still possible (by calling `.save()` but not called automatically upon stopping

'chnl_in' [microphone channel;] see `psychopy.sound.backend.get_input_devices()`

'chnl_out': not implemented; output device to use

'start': 0, select section from a file based on (start, stop) time

'stop': -1, end of file (default)

'vol': 0.99, volume 0..1

'low': 100, Hz, low end of bandpass; can vary for M/F speakers

'high': 3000, Hz, high end of bandpass

'threshold': 10

'baseline': 0; 0 = auto-detect; give a non-zero value to use that

'more_processing': True; compute more stats per chunk including bandpass; try False if 32-bit python can't keep up

'zero_crossings': True

`_do_chunk ()`

Core function to handle a chunk (= a few ms) of input.

There can be small temporal gaps between or within chunks, i.e., *slippage*. Adjust several parameters until this is small: `msPerChunk`, and what processing is done within `._process()`.

A trigger (`_chunktrig`) signals that `_chunktable` has been filled and has set `_do_chunk` as the function to call upon triggering. `.play()` the trigger again to start recording the next chunk.

`_process (chunk)`

Calculate and store basic stats about the current chunk.

This gets called every chunk – keep it efficient, esp 32-bit python

`_set_baseline ()`

Set `self.baseline` = rms(silent period) using `_baselinetable` data.

Called automatically (via pyo trigger) when the baseline table is full. This is better than using chunks (which have gaps between them) or the whole table (which can be very large = slow to work with).

`_set_defaults ()`

Set remaining defaults, initialize lists to hold summary stats

`_set_signaler ()`

Set the signaler to be called by `trip()`

`_set_source ()`

Data source: `file_in`, array, or microphone

`_set_tables ()`

Set up the pyo tables (allocate memory, etc).

One source -> three pyo tables: `chunk=short`, `whole=all`, `baseline`. triggers fill tables from `self._source`; make triggers in `.start()`

`detect ()`

Trip if recent audio power is greater than the baseline.

`join (sec=None)`

Sleep for `sec` or until end-of-input, and then call `stop()`.

save (*ftype=""*, *dtype='int16'*)

Save new data to file, return the size of the saved file (or None).

The file format is inferred from the filename extension, e.g., *flac*. This will be overridden by the *ftype* if one is provided; defaults to *wav* if nothing else seems reasonable. The optional *dtype* (e.g., *int16*) can be any of the sample types supported by *pyo*.

property slippage

Ratio of the actual (elapsed) time to the ideal time.

Ideal ratio = 1 = sample-perfect acquisition of *msPerChunk*, without any gaps between or within chunks. 1. / *slippage* is the proportion of samples contributing to chunk stats.

Type Diagnostic

start (*silent=False*)

Start reading and processing audio data from a file or microphone.

property started

Boolean property, whether *.start()* has been called.

stop ()

Stop a voice-key in progress.

Ends and saves the recording if using microphone input.

wait_for_event (*plus=0*)

Start, join, and wait until the voice-key trips, or it times out.

Optionally wait for some extra time, *plus*, before calling *stop()*.

class `psychopy.voicekey.OffsetVoiceKey` (*sec=10*, *file_out=""*, *file_in=""*, *delay=0.3*, ***kwargs*)

Class to detect the offset of a single-word utterance.

Record and ends the recording after speech offset. When the voice key trips, the best voice-offset RT estimate is saved as *self.event_offset*, in seconds.

Parameters

sec: duration of recording in the absence of speech or other sounds.

delay: extra time to record after speech offset, default 0.3s.

The same methods are available as for class `OnsetVoiceKey`.

9.27.3 Signal-processing functions

Several utility functions are available for real-time sound analysis.

`psychopy.voicekey.smooth` (*data*, *win=16*, *tile=True*)

Running smoothed average, via convolution over *win* window-size.

tile with the mean at start and end by default; otherwise replace with 0.

`psychopy.voicekey.bandpass` (*data*, *low=80*, *high=1200*, *rate=44100*, *order=6*)

Return bandpass filtered *data*.

`psychopy.voicekey.rms` (*data*)

Basic audio-power measure: root-mean-square of data.

Identical to *std* when the mean is zero; faster to compute just rms.

`psychopy.voicekey.std` (*data*)

Like rms, but also subtracts the mean (= slower).

`psychoPy.voicekey.zero_crossings` (*data*)

Return a vector of length *n*-1 of zero-crossings within vector *data*.

1 if the adjacent values switched sign, or 0 if they stayed the same sign.

`psychoPy.voicekey.tone` (*freq=440, sec=2, rate=44100, vol=0.99*)

Return a `np.array` suitable for use as a tone (pure sine wave).

`psychoPy.voicekey.apodize` (*data, ms=5, rate=44100*)

Apply a Hanning window (5ms) to reduce a sound's 'click' onset / offset.

9.27.4 Sound file I/O

Several helper functions are available for converting and saving sound data from several data formats (numpy arrays, pyo tables) and file formats. All file formats that *pyo* supports are available, including *wav*, *flac* for lossless compression. *mp3* format is not supported (but you can convert to *.wav* using another utility).

`psychoPy.voicekey.samples_from_table` (*table, start=0, stop=-1, rate=44100*)

Return samples as a `np.array` read from a pyo table.

A (start, stop) selection in seconds may require a non-default rate.

`psychoPy.voicekey.table_from_samples` (*samples, start=0, stop=-1, rate=44100*)

Return a pyo `DataTable` constructed from samples.

A (start, stop) selection in seconds may require a non-default rate.

`psychoPy.voicekey.table_from_file` (*file_in, start=0, stop=-1*)

Read data from files, any pyo format, returns (rate, pyo `SndTable`)

`psychoPy.voicekey.samples_from_file` (*file_in, start=0, stop=-1*)

Read data from files, returns tuple (rate, `np.array(.float64)`)

`psychoPy.voicekey.samples_to_file` (*samples, rate, file_out, fmt="", dtype='int16'*)

Write data to file, using requested format or infer from file *.ext*.

Only integer *rate* values are supported.

See <http://ajaxsoundstudio.com/pyodoc/api/functions/sndfile.html>

`psychoPy.voicekey.table_to_file` (*table, file_out, fmt="", dtype='int16'*)

Write data to file, using requested format or infer from file *.ext*.

9.28 psychoPy.web - Web methods

9.28.1 Test for access

`psychoPy.web.haveInternetAccess` (*forceCheck=False*)

Detect active internet connection or fail quickly.

If *forceCheck* is *False*, will rely on a cached value if possible.

`psychoPy.web.requireInternetAccess` (*forceCheck=False*)

Checks for access to the internet, raise error if no access.

9.28.2 Proxy set-up and testing

`psychopy.web.setupProxy` (*log=True*)

Set up the urllib proxy if possible.

The function will use the following methods in order to try and determine proxies:

1. standard `urllib.request.urlopen` (which will use any statically-defined http-proxy settings)
2. previous stored proxy address (in prefs)
3. proxy.pac files if these have been added to system settings
4. auto-detect proxy settings (WPAD technology)

Returns

Return type True (success) or False (failure)

Further information:

TIMING INFORMATION FOR PSYCHOPY

There is documentation about how to optimize timing in at *Timing Issues and synchronisation*

We recently ran a study [testing the timing on a wide range of software packages](#), online and offline. The data for that study are available below:

10.1 Mega-timing study data

Here are the data summaries for our paper, [The timing mega-study: comparing a range of experiment generators, both lab-based and online](#)

You can read the full preprint of the paper at <https://peerj.com/articles/9414/>

10.1.1 Table2: lab-based timing data

This is a sortable version of Table2 from [The timing mega-study: comparing a range of experiment generators, both lab-based and online](#)

Timing summaries of lab-based software by package and platform. The Var(iability) measures are the inter-trial standard deviations of the various latencies for that configuration. The table is sorted by the mean of those variabilities (Mean Var). The Lag/Bias measures are the mean latency values, for that configuration. In the case of audiovisual sync, a negative bias indicates the audio lead the visual stimulus, a positive bias means the visual lead the audio. Each of the values with a hyperlink will lead to a plot of the distribution of values leading to that summary value.

10.1.2 Table3: online timing data

This is a sortable version of Table3 from [The timing mega-study: comparing a range of experiment generators, both lab-based and online](#)

Timing summaries of web-based software by package, platform, and browser. The Var(iability) measures are the inter-trial standard deviations of the various latencies for that configuration. The table is sorted by the mean of those variabilities (Mean Var). The Lag/Bias measures are the mean latency values, for that configuration. In the case of audiovisual sync, a negative bias indicates the audio lead the visual stimulus, a positive bias means the visual lead the audio. Each of the values with a hyperlink will lead to a plot of the distribution of values leading to that summary value.

TROUBLESHOOTING

Regrettably, occasionally has bugs. Running on all possible hardware and all platforms is a big ask. That said, a huge number of bugs have been resolved by the fact that there are literally 1000s of people using the software that have *contributed either bug reports and/or fixes*.

Below are some of the more common problems and their workarounds, as well as advice on how to get further help.

11.1 The application doesn't start

You may find that you try to launch the application, the splash screen appears and then goes away and nothing more happens. What this means is that an error has occurred during startup itself.

Commonly, the problem is that a preferences file is somehow corrupt. To fix that see *Cleaning preferences and app data*, below.

If resetting the preferences files doesn't help then we need to get to an error message in order to work out why the application isn't starting. The way to get that message depends on the platform (see below).

Windows users (starting from the Command Prompt):

1. Did you get an error message that "This application failed to start because the application configuration is incorrect. Reinstalling the application may fix the problem"? If so that indicates you need to [update your .NET installation to SP1](#).
2. **open a Command Prompt (terminal):**
 1. go to the Windows Start menu
 2. select Run... and type in cmd <Return>
3. paste the following into that window (Ctrl-V doesn't work in Cmd.exe but you can right-click and select Paste):

```
"C:\Program Files\PsychoPy2\python.exe" -m psychopy.app.psychopyApp
```

4. when you hit <return> you will hopefully get a moderately useful error message that you can *Contribute to the Forum (mailing list)*

Mac users:

1. open the Console app (open spotlight and type console)
2. if there are a huge number of messages there you might find it easiest to clear them (the brush icon) and then start again to generate a new set of messages

11.2 I run a Builder experiment and nothing happens

An error message may have appeared in a dialog box that is hidden (look to see if you have other open windows somewhere).

An error message may have been generated that was sent to output of the Coder view:

1. go to the Coder view (from the Builder>View menu if not visible)
2. if there is no Output panel at the bottom of the window, go to the View menu and select Output
3. try running your experiment again and see if an error message appears in this Output view

If you still don't get an error message but the application still doesn't start then manually turn off the viewing of the Output (as below) and try the above again.

11.3 Manually turn off the viewing of output

Very occasionally an error will occur that crashes the application *after* the application has opened the Coder Output window. In this case the error message is still not sent to the console or command prompt.

To turn off the Output view so that error messages are sent to the command prompt/terminal on startup, open your appData.cfg file (see *Cleaning preferences and app data*), find the entry:

```
[coder]
showOutput = True
```

and set it to *showOutput = False* (note the capital 'F').

11.4 Use the source (Luke?)

comes with all the source code included. You may not think you're much of a programmer, but have a go at reading the code. You might find you understand more of it than you think!

To have a look at the source code do one of the following:

- when you get an error message in the *Coder* click on the hyperlinked error lines to see the relevant code
- **on Windows**
 - go to *<location of PsychoPy app>\Lib\site-packages\psychopy*
 - have a look at some of the files there
- **on Mac**
 - right click the app and select *Show Package Contents*
 - navigate to *Contents/Resources/lib/pythonX.X/psychopy*

11.5 Cleaning preferences and app data

Every time you shut down (by normal means) your current preferences and the state of the application (the location and state of the windows) are saved to disk. If is crashing during startup you may need to edit those files or delete them completely.

The exact location of those files varies by machine but on windows it will be something like `%APPDATA%\psychopy3` and on Linux/MacOS it will be something like `~/psychopy3`. You can find it running this in the commandline (if you have multiple Python installations then make sure you change `python` to the appropriate one for :

```
python -c "from psychopy import prefs; print(prefs.paths['userPrefsDir'])"
```

Within that folder you will find `userPrefs.cfg` and `appData.cfg`. The files are simple text, which you should be able to edit in any text editor.

If the problem is that you have a corrupt experiment file or script that is trying and failing to load on startup, you could simply delete the `appData.cfg` file. Please *also Contribute to the Forum (mailing list)* a copy of the file that isn't working so that the underlying cause of the problem can be investigated (google first to see if it's a known issue).

11.6 Errors with getting/setting the Gamma ramp

There are two common causes for errors getting/setting gamma ramps depending on whether you're running Windows or Linux (we haven't seen these problems on Mac).

11.6.1 MS Windows bug in release 1903

In Windows release 1903 Microsoft added a [bug that prevents getting/setting the gamma ramp](#). This only occurs in certain scenarios, like when the screen orientation is in portrait, or when it is extended onto a second monitor, but it does affect **all versions of PsychoPy**.

For the Windows bug the workarounds are as follows:

If you don't need gamma correction then, as of 3.2.4, you can go to the preferences and set the `defaultGammaFailPolicy` to be 'warn' (rather than 'abort') and then your experiment will still at least run, just without gamma correction.

If you do need gamma correction then there isn't much that the team can do until Microsoft fixes the underlying bug. You'll need to do one of:

- Not using Window 1903 (e.g. revert the update) until a fix is listed on the [status of the gamma bug](#)
- Altering your monitor settings in Windows (e.g. turning off extended desktop) until it works . Unfortunately that might mean you can't use dual independent displays for vision science studies until Microsoft fix it.

11.6.2 Linux missing xorg.conf

On Linux some systems appear to be missing a configuration file and adding this back in and restarting should fix things.

Create the following file (including the folders as needed):

```
/etc/X11/xorg.conf.d/20-intel.conf
```

and put the following text inside (assuming you have an intel card, which is where we've typically seen the issue crop up):

```
Section "Device"  
    Identifier "Intel Graphics"  
    Driver "intel"  
EndSection
```

For further information on the discussion of this (Linux) issue see <https://github.com/psychopy/psychopy/issues/2061>

ALERTS

The Alerts system is designed to provide you information of varying levels about things that may be a concern in your study (but equally may not be - it depends on your study!). Alerts show up within with a code and that code should take you here for detailed information about the root cause of the problem, the workarounds, and the type of study that might be badly affected by this issue.

12.1 2xxx: Issues with units

These issue usually result in problems with stimuli not appearing (too fast, too small, off-screen) or being an unexpected size or color.

12.1.1 2115: Stimulus size is bigger than the window dimensions

Synopsis

Your stimulus size exceeds the X or Y window dimensions. Stimuli sized greater than the window size will not be completely visible.

Details

This issue is often caused by an inconsistency between the units of your stimulus and the values being requested. For instance a size of 3, when the units are *deg* is sensible (3 degrees would be within the screen dimensions) but a size of 3 when the units are *height* would not be sensible (3 times bigger than the height of the screen).

PsychoPy versions affected

All versions.

Solutions

Check the size and the *units of the stimulus* carefully. You may also need to check the monitor calibration if you're using units that depend on the monitor size and resolution (like *cm* and *deg*).

12.1.2 2120: Stimulus size is smaller than 1 pixel

Synopsis

Stimuli size requested is smaller than 1 pixel on X and/or Y dimensions. Stimuli sized smaller than 1 pixel will not be visible.

Details

This issue is often caused by an inconsistency between the units of your stimulus and the values being requested. For instance a size of 0.1, when the units are *height* is sensible (1/10th the height of the screen) but a size of 0.1 when the units are *pix* would not be sensible (1/10th of a pixel).

PsychoPy versions affected

All versions

Solutions

Check the size and the *units of the stimulus* carefully. You may also need to check the monitor calibration if you're using units that depend on the monitor size and resolution (like *cm* and *deg*).

12.1.3 2155: Stimulus position is beyond the bounds of the window

Synopsis

Your stimulus position exceeds the X or Y window dimensions. Stimuli centered beyond the window will not be completely visible.

Details

This issue is often caused by an inconsistency between the units of your stimulus and the values being requested. For instance a position of (3, 0), when the units are *deg* is sensible (3 degrees to the right of the screen center) but a position of (3, 0) when the units are *height* would not be sensible (3 times the height of the screen to the right of the center).

PsychoPy versions affected

All versions

Solutions

Check the position and the *units of the stimulus* carefully. You may also need to check the monitor calibration if you're using units that depend on the monitor size and resolution (like *cm* and *deg*).

12.2 3xxx: Issues with timing

There are many ways for timing to go wrong, and these alerts will warn you about some of them (but we do always recommend that you test your timing carefully with hardware devices if precise timing is important to your study).

12.2.1 3110: Stimulus duration is less than one screen refresh

Synopsis

Your stimulus is scheduled to last for a duration that can't be achieved with a normal 60 Hz monitor. Duration will implicitly be round up (probably) to the next frame duration.

Requested start or stop times of visual components cannot be presented for times requested. Accurate presentation times must be in increments of your screen refresh rate.

Details

Stimuli can only be presented for a fixed number of screen refreshes. If your screen has a refresh rate of 60 Hz (common for standard monitors) then each screen refresh period lasts 1/60 s (roughly 16.6667 ms). That means you can present your stimulus for 16.7 ms but not for, say, 5 ms because that would require the stimulus to be presented for half of one screen refresh period.

PsychoPy versions affected

All versions.

Solutions

We recommend for brief stimuli that you simply specify your stimulus duration in terms of the number of frames it should be presented (e.g. 1, 2, 3, for 16.7, 33.3 and 50 ms respectively). That reminds you of what is possible and means that PsychoPy won't have to guess about what to do when the desired duration isn't achievable.

If you need stimuli to be presented for briefer durations than 16.7 ms then you should look into high-frame-rate displays (100, 120 and 144 Hz displays are all available). There are also now variable-frame-rate monitors that can vary the duration of each frame within limits. If you are *already* using a high- or variable-rate monitor then this alert may not be relevant to you.

12.2.2 3115: Stimulus duration is not possible on a standard monitor refresh

Synopsis

If using a 60Hz or 100Hz monitor, then for accurate presentation of visual stimuli, components must be presented in valid multiples of screen refresh for 60 Hz or 100 Hz.

Details

When presenting stimuli at, say, 60 Hz you stimulus can be presented for 1 frame ($1/60 \text{ s} = 16.667 \text{ ms}$), 2 frames ($2/60 \text{ s} = 33.333 \text{ ms}$) but not for intervening periods (20 ms is not possible because the stimulus would have to be presented for a little more than 1 frame, which isn't physically possible on standard fixed-framerate monitors).

PsychoPy versions affected

All versions.

Solutions

We recommend for brief stimuli that you simply specify your stimulus duration in terms of the number of frames it should be presented (e.g. 1, 2, 3, for 16.7, 33.3 and 50 ms respectively). That reminds you of what is possible and means that PsychoPy won't have to guess about what to do when the desired duration isn't achievable.

If you need stimuli to be presented for briefer durations than 16.7 ms then you should look into high-frame-rate displays (100, 120 and 144 Hz displays are all available). There are also now variable-frame-rate monitors that can vary the duration of each frame within limits. If you are *already* using a high- or variable-rate monitor then this alert may not be relevant to you.

12.3 4xxx: Issues with Builder experiments

Issues specifically around Builder experiments doing unexpected things.

12.3.1 4051: Experiment from future version

Synopsis

It looks like you're trying to open an experiment built in a newer version of PsychoPy than you currently have installed. This can cause problems, your experiment may run differently to how you expect or may not run at all.

Details

Between different version of PsychoPy, we make a number of changes to improve usability and performance, but which mean that newer experiments may contain code which older versions do not know how to handle. We always try to maintain "backwards compatibility" - so that experiments built in older versions run the same in newer versions. However, we cannot predict what changes we will make years down the line, so cannot guarantee "forwards compatibility" in the same way. This means that experiments built on newer versions of PsychoPy may run differently on older versions, or may not run at all.

PsychoPy versions affected

Any

Solutions

To fix this, we recommend updating to the newest version of PsychoPy. Or, if you need an older version for other reasons, you can set this specific experiment to run in a different version by changing the “Use Version” parameter in Experiment Settings to the version it was created in (or a newer version).

12.3.2 4052: Experiment fixed to past version

Synopsis

It looks like your experiment is set to run in a version before 2021.1.0, the version of PsychoPy you have currently installed is newer than this, so saving the experiment in your current version may add new types of parameters which the version it is set to cannot interpret.

Details

Between different version of PsychoPy, we make a number of changes to improve usability and performance, but which mean that newer experiments may contain code which older versions do not know how to handle. We always try to maintain “backwards compatibility” - so that experiments built in older versions run the same in newer versions. However, we cannot predict what changes we will make years down the line, so cannot guarantee “forwards compatibility” in the same way. This means that experiments built in the current version of PsychoPy but set to run in an older version may not run as expected, or at all. This is a particular issue between 2020.2.10 and 2021.1.0 as between the two we added several new types of parameter, meaning if you run in 2020.2 PsychoPy will not recognise them.

PsychoPy versions affected

>2021.1.0

Solutions

To fix this, we recommend setting the experiment version in the Experiment Settings menu to be 2021.1.0 or newer, as these will have the ability to recognise the new parameter types.

12.3.3 4105: Component start time exceeds its stop time

Synopsis

A component start time/frame exceeds the stop time/frame. This means that the component starts *after* it is due to finish and the stimulus will most likely not be shown at all.

PsychoPy versions affected

All versions.

Solutions

Check your start and stop time carefully, including the units being used. For example your stimulus might be set to start at a certain *time* (say 36 seconds) rather than frame number 36.

12.3.4 4115: Component start/stop in units of frames must be whole numbers

Synopsis

Component start and stop times/durations in frames must be given as whole numbers.

Details

Since it isn't possible to start or stop a stimulus part-way through a screen refresh it would be unwise to request that PsychoPy attempts that.

PsychoPy versions affected

All versions.

Solutions

Check your stimulus start/stop time and set to an integer value instead of a decimal value.

12.3.5 4120: Component stop duration with no start time

Synopsis

In order for stop time to be set as a duration, PsychoPy needs to know the time at which the component started. When there's no start time, stop time will be calculated from the duration as if start time were 0s.

Details

PsychoPy versions affected

> 2022.1.0

Solutions

You can either add a start time to this component, or change the stop time to be set in a different way (e.g. *time (s)*)

12.3.6 4205: Probable syntax error detected in your Python code

Synopsis

Python syntax error found in your code component.

Details

This *may* be spurious in that the code check may have failed to understand something that is a valid syntax (syntax in Python can change according to version) but this will become clear if you run your experiment and it fails to run.

PsychoPy versions affected

All versions.

Solutions

Check the code carefully at the indicated line and on the few lines above. Check especially for things like un-matched parentheses or quote symbols.

12.3.7 4210: Probable syntax error detected in your JavaScript code

Synopsis

JavaScript syntax error found in your code component.

Details

This alert *may* be spurious in that the code check may have failed to understand something that is a valid syntax (syntax in JavaScript can change according to version) but this will become clear if you run your experiment and it fails to run.

PsychoPy versions affected

All versions.

Solutions

Check the code carefully at the indicated line and on the few lines above. Check especially for things like un-matched parentheses or quote symbols.

12.3.8 4305: Component is currently disabled in your experiment

Synopsis

You have a disabled Component in your experiment. This alert is created to inform users that disabled components will not be written to your experiment script. This may not be intentional, and will therefore affect your desired outcome.

Details

Most likely you disabled the Component deliberately while testing things out, but this Alert is making sure you remember that it isn't currently operational.

PsychoPy versions affected

>= 3.1.0

Solutions

Re-enable the Component in the Builder view (component dialog boxes each have a “testing” tab: unselect the “Disable component” setting there). Otherwise just ignore this Alert if you intended it to be disabled.

12.3.9 4310: Builder cannot check your parameter further

Synopsis

This alert is received when an integrity check has failed to calculate a parameter. Most commonly, this is because a variable from code, or a conditions file, has been encountered, and the value of the variable is not available to the integrity checking system.

Details

Longer description with optional links to further information

PsychoPy versions affected

>= 3.2.3

Solutions

This alert is for information only, and does not require any action.

12.3.10 4315: Invalid dollar sign syntax.

Synopsis

This alert is received when a dollar sign has been used incorrectly in a stimulus parameter.

Details

By putting a dollar sign at the beginning of a parameter value, you can indicate that the parameter should be interpreted as code rather than as a string. However, a dollar sign should not appear anywhere else in the parameter value, unless it is either: - After a # when the parameter is interpreted as code (meaning it will be commented out) - Immediately after a ```, an escape character (escaped dollar signs are only valid when the parameter is not interpreted as code, or within quotation marks if it is)

PsychoPy versions affected

All

Solutions

If the dollar sign is a part of a string, you should add an escape character (```) *immediately before it*. *If it is supposed to be in a comment, it must appear after a `#`*. Otherwise, remove any dollar signs which are not at the very beginning of the value.

12.3.11 4320: Non-local font.

Synopsis

This alert is received when a font has been specified which PsychoPy cannot find on your local machine.

Details

Google Fonts is a repository of thousands of free fonts, which PsychoPy is able to access on-the-fly, allowing you to use any font within the Google Fonts library (fonts.google.com) as if it were installed on your machine. However, retrieving this font requires PsychoPy to connect to the internet and send/receive some data, so we raised this alert to give you some warning that this will happen. Font files are tiny, so in most cases this will not be noticeable, however if you are on a strictly metered connection or you are not connected to the internet at all then this may cause some issues.

PsychoPy versions affected

All

Solutions

If your computer is connected to the internet and you don't have any limits on how much data you can send/receive, then no action is needed! PsychoPy will happily go off and find this font for you. However, if you are running offline or you have strict data limits, then you should install this font locally instead. Google Fonts can be downloaded for free from fonts.google.com as .ttf files, once downloaded these can be copied to a memory stick and installed on any machine you need by simply opening the file and clicking "Install".

12.3.12 4325: Font not available

Synopsis

The font you requested could not be found in the weight and style you requested, this alert is to warn you that your component will use Open Sans Regular (fonts.google.com/specimen/Open+Sans)

Details

There are a few reasons why you might be receiving this alert: 1. The font you requested does not exist at all, there may be a typo in the font name. 2. The font is one which is not installed on your local machine or available on Google Fonts. 3. The font you requested exists, but not in the style you requested (bold, italic, etc.). For example, if you requested the font *Raleway Dots* (fonts.google.com/specimen/Raleway+Dots) in bold, you will always receive this error as this font only exists in regular. 4. The font you requested exists on Google Fonts, but could not be retrieved, for example if you are connected to the internet.

PsychoPy versions affected

All

Solutions

The first things to check are that the name of the font is spelled correctly, that the font exists and that you are connected to the internet. You can check which fonts are installed on your machine through the Settings for your operating system, you can check whether the font exists on Google Fonts by going to fonts.google.com and searching for it. If the font exists, you can check (either on Google Fonts or your operating system's font manager) what weights and styles it exists in. If *bold* is ticked in Builder, then the requested font weight is 700 (Bold), if not then the requested weight is 400 (Regular). You can set the font weight more precisely by supplying a numeric font weight to the component rather than just True or False, for example if you wanted the font to be Extra-Light you could supply the value 200.

12.3.13 4330: Recording device not found

Synopsis

The recording device specified in your Microphone component could not be found on your current machine, so when writing the Python code for this experiment to run, the default device will be used instead.

Details

When writing Python code for Microphone components, PsychoPy needs to know the numeric index of its recording device. If the device isn't connected, then this information isn't available, so the Python code is written with the default device index instead.

PsychoPy versions affected

> 2021.2.0

Solutions

The information in your *.psyexp* file won't be changed, just the compiled Python file. If you open the same *.psyexp* file on a machine with the device connected, the Python code generated will contain that device's index, so if you're just testing the experiment on a different machine to the one it will run on then you can ignore this alert.

12.3.14 4335: Component or routine not implemented in Python

Synopsis

This component is not yet implemented in Python, meaning that it will do nothing in local experiments.

Details

You are receiving this alert as you are compiling an experiment to JavaScript, but the experiment contains a component which only works in JavaScript.

PsychoPy versions affected

> 2022.1.0

Solutions

No action required.

12.3.15 4340: Component or routine not implemented in JavaScript

Synopsis

This component is not yet implemented in JavaScript, meaning that it will do nothing in online experiments.

Details

You are receiving this alert as you are compiling an experiment to JavaScript, but the experiment contains a component which only works in Python.

PsychoPy versions affected

> 2022.1.0

Solutions

No action required.

12.3.16 4405: Textbox and keyboard conflict

Synopsis

As editable Textbox components and Keyboard components both listen for key presses, the two can often come into conflict and cause problems.

Details

As editable Textbox components and Keyboard components both listen for key presses, the two can often come into conflict and cause problems. For example, if a Keyboard component is listening for *Enter* to end the routine, what happens when the participant presses *Enter* to start a new line?

PsychoPy versions affected

> 2020.1.0

Solutions

In general, we recommend ending routines containing editable textboxes using a Button, Mouse or ROI component rather than Keyboard. If you are trying to gather keyboard responses as well as text input, consider splitting the two into separate routines to avoid conflicts.

12.3.17 4505: Eyetracking not configured

Synopsis

Experiment includes components or routines which use eyetracking, but no eye tracker is configured.

Details

In order for an Eyetracker Record, Eyetracker Calibrate or Eyetracker Validate routine to work, there needs to be an eyetracker set up in Experiment Settings.

PsychoPy versions affected

>2021.2

Solutions

Either remove any eyetracking components and routines from the experiment flow or set up an eyetracker in the Eyetracker tab of Experiment Settings. If you are testing an eye tracking experiment but do not have an eye tracker connected, you can use MouseGaze to simulate eye movements with the mouse.

12.3.18 4510: Eyetracker not calibrated

Synopsis

The experiment is set up to use an eye tracker, but there is no calibration routine in the experiment flow.

Details

In order to get accurate readings, an eye tracker needs to know what points on the screen correspond to what eye movements. It learns this during a “calibration” routine - in which the participant looks at different points on the screen whose positions are known to the eye tracker. While an eyetracker can sometimes guess, readings will be much more accurate if the eyetracker has been calibrated.

PsychoPy versions affected

>2021.2

Solutions

Either set the *Eyetracker* setting in *Experiment Settings* to be *None* (no eye tracking) or *MouseGaze* (use the mouse as if it were an eye tracker) so that no calibration is needed, or add a calibration routine to the experiment flow. You can find the button to create a calibration routine in the Eyetracking section of the Components panel and can add it to the flow via the Add Routine button.

12.3.19 4520: Animation params not used

Synopsis

Some eyetrackers do not support animations between target stimuli in their calibration routine, so although you have enabled animation in your calibration routine, due to the limitations of the eyetracker you are using any settings for animation will not be used.

Details

PsychoPy versions affected

> 2021.2.0

Solutions

If you are only using the current eyetracker to test an experiment and will be using one which supports target animations, then you don't need to do anything. This alert will stop appearing when you use the other eyetracker.

Otherwise, you just need to be aware that the settings you've specified for animation will not have any effect.

12.3.20 4530: Auto pace param not used

Synopsis

Some eyetrackers do not support manual pacing for calibration, as pacing is always handled automatically. If you have set "Auto-Pacing" to False while using one of these eyetrackers, this setting will be ignored.

Details

PsychoPy versions affected

> 2021.2.0

Solutions

If you are only using the current eyetracker to test an experiment and will be using one which supports manual pacing, then you don't need to do anything. This alert will stop appearing when you use the other eyetracker.

Otherwise, you just need to be aware that pacing will be automatic.

12.3.21 4540: Eyetracking requires window to be fullscreen

Synopsis

Eyetracking requires the experiment window to be fullscreen, otherwise the coordinates returned by the eyetracker won't line up with the coordinates of stimuli within your experiment

Details

PsychoPy versions affected

> 2021.2.0

Solutions

In Experiment Settings, make sure that "Full-screen window" is checked.

12.3.22 4545: Eyetracking requires a monitor config

Synopsis

In order for eyetracking measurements to be accurate, the eyetracker needs to know about the monitor you are using - how wide is it? How tall? What is its resolution? This is so that it can translate eye movements (in degrees of visual angle) into usable data (in height, pix, etc. units). Without this information, the eyetracker can only guess and will therefore give only approximate data.

Details

PsychoPy versions affected

> 2021.2.0

Solutions

Using the Monitor Centre, configure your monitor settings.

12.3.23 4550: Input -> Keyboard Backend not set to 'ioHub'

Synopsis

Experiment is configured to use an eye tracker but Input -> Keyboard Backend experiment setting is set to 'Psych-Toolbox'.

Details

Eye trackers run using ioHub, which also handles Keyboard events, so Input -> Keyboard Backend experiment setting should be 'ioHub'.

PsychoPy versions affected

>=2022.1

Solutions

Switch the Input -> Keyboard Backend experiment setting to use 'ioHub'

12.3.24 4605: Transcription service not compatible online

Synopsis

Some audio transcription services can only be run online, some can only be run locally and some can be run either way.

If you are receiving this alert, it means that the audio transcription service you selected is one which either only works locally, but you are trying to run your experiment online, or it is not supported at all.

Details

PsychoPy versions affected

>2021.2.0

Solutions

To silence this alert, choose a transcription service which can be run online, such as Google or Azure.

12.3.25 4610: Transcription service not compatible locally

Synopsis

Some audio transcription services can only be run online, some can only be run locally and some can be run either way.

If you are receiving this alert, it means that the audio transcription service you selected is one which only works online, but you are trying to run your experiment locally, or it is not supported at all.

Details

PsychoPy versions affected

>2021.2.0

Solutions

To silence this alert, choose a transcription service which can be run online, such as the built-in transcriber.

12.3.26 4615: API key not found

Synopsis

If you're receiving this alert, it means you have selected an audio transcriber which requires an API key to function but have not supplied an API key in preferences.

Details

Some audio transcribers work fine without a key, they're just a publicly available Python or JavaScript function which runs using your own computer's processing power, but others use algorithms which are either confidential or require huge amounts of processing power. This means that they need to be run on a server, often one which you've paid to access. If you have such a subscription, then your chosen transcription service will have provided you with an "API key" - this is a unique code, like a password, which tells their server who you are. If you are using such a transcription service within PsychoPy, then PsychoPy needs to be able to use your API key to request transcription.

PsychoPy versions affected

> 2021.2.0

Solutions

In PsychoPy, go to File -> Preferences -> General. Here you will find some preferences starting with *transcrKey* - in the one matching your choice of transcription service, copy and paste the API key you were given when you subscribed to that service.

RECIPES (“HOW-TO”S)

Below are various tips/tricks/recipes/how-tos for PsychoPy. They involve something that is a little more involved than you would find in FAQs, but too specific for the manual as such (should they be there?).

13.1 Adding external modules to Standalone PsychoPy

You might find that you want to add some additional Python module/package to your Standalone version of PsychoPy. To do this you need to:

- download a copy of the package (make sure it’s for Python 2.7 on your particular platform)
- unzip/open it into a folder
- add that folder to the path of PsychoPy by one of the methods below

Avoid adding the entire path (e.g. the site-packages folder) of separate installation of Python, because that may contain conflicting copies of modules that PsychoPy is also providing.

13.1.1 Using preferences

As of version 1.70.00 you can do this using the PsychoPy preferences/general. There you will find a preference for *paths* which can be set to a list of strings e.g. `['/Users/jwp/code', '~/code/thirdParty']`

These only get added to the Python path when you import psychopy (or one of the psychopy packages) in your script.

13.1.2 Adding a .pth file

An alternative is to add a file into the site-packages folder of your application. This file should be pure text and have the extension .pth to indicate to Python that it adds to the path.

On win32 the site-packages folder will be something like *C:/Program Files/PsychoPy2/lib/site-packages*

On macOS you need to right-click the application icon, select ‘Show Package Contents’ and then navigate down to Contents/Resources/lib/pythonX.X. Put your .pth file here, next to the various libraries.

The advantage of this method is that you don’t need to do the import psychopy step. The downside is that when you update PsychoPy to a new major release you’ll need to repeat this step (patch updates won’t affect it though).

13.2 Animation

General question: How can I animate something?

Conceptually, animation just means that you vary some aspect of the stimulus over time. So the key idea is to draw something slightly different on each frame. This is how movies work, and the same principle can be used to create scrolling text, or fade-in / fade-out effects, and the like.

(copied & pasted from the email list; see the list for people's names and a working script.)

13.3 Scrolling text

Key idea: Vary the **position** of the stimulus across frames.

Question: How can I produce scrolling text (like html's <marquee behavior = "scroll" > directive)?

Answer: PsychoPy has animation capabilities built-in (it can even produce and export movies itself (e.g. if you want to show your stimuli in presentations)). But here you just want to animate stimuli directly.

e.g. create a text stimulus. In the 'pos' (position) field, type `[frameN, 0]`

and select "set every frame" in the popup button next to that field.

Push the Run button and your text will move from left to right, at one pixel per screen refresh, but stay at a fixed y-coordinate. In essence, you can enter an arbitrary formula in the position field and the stimulus will be-redrawn at a new position on each frame. `frameN` here refers to the number of frames shown so far, and you can extend the formula to produce what you need.

You might find performance issues (jittering motion) if you try to render a lot of text in one go, in which case you may have to switch to using images of text.

I wanted my text to scroll from right to left. So if you keep your eyes in the middle of the screen the next word to read would come from the right (as if you were actually reading text). The original formula posted above scrolls the other way. So, you have to put a negative sign in front of the formula for it to scroll the other way. You have to change the units to pixel. Also, you have to make sure you have an end time set, otherwise it just flickers. I also set my letter height to 100 pixels. The other problem I had was that I wanted the text to start blank and scroll into the screen. So, I wrote `[2000-frameN, 0]`

and this worked really well.

13.4 Fade-in / fade-out effects

Key idea: vary the **opacity** of the stimulus over frames.

Question: I'd like to present an image with the image appearing progressively and disappearing progressively too. How to do that?

Answer: The Patch stimulus has an opacity field. Set the button next to it to be "set every frame" so that its value can be changed progressively, and enter an equation in the box that does what you want.

e.g. if your screen refresh rate is 60 Hz, then entering `frameN/120`

would cycle the opacity linearly from 0 to 1.0 over 2s (it will then continue incrementing but it doesn't seem to matter if the value exceeds 1.0).

Using a code component might allow you to do more sophisticated things (e.g. fade in for a while, hold it, then fade out). Or more simply, you just create multiple successive Patch stimulus components, each with a different equation or value in the opacity field depending on their place in the timeline.

13.5 Building an application from your script

A lot of people ask how they can build a standalone application from their Python script. Usually this is because they have a collaborator and want to just send them the experiment.

In general this is not advisable - the resulting bundle of files (single file on macOS) will be on the order of 100Mb and will not provide the end user with any of the options that they might need to control the task (for example, Monitor Center won't be provided so they can't to calibrate their monitor). A better approach in general is to get your collaborator to install the Standalone PsychoPy on their own machine, open your script and press run. (You don't send a copy of Microsoft Word when you send someone a document - you expect the reader to install it themselves and open the document).

Nonetheless, it is technically possible to create exe files on Windows, and Ricky Savjani (savjani at bcm.edu) has kindly provided the following instructions for how to do it. A similar process might be possible on macOS using py2app - if you've done that then feel free to contribute the necessary script or instructions.

13.5.1 Using py2exe to build an executable

Instructions:

1. Download and install py2exe (<http://www.py2exe.org/>)
2. Develop your PsychoPy script as normal
3. Copy this setup.py file into the same directory as your script
4. Change the Name of progName variable in this file to the Name of your desired executable program name
5. **Use cmd (or bash, terminal, etc.) and run the following in the directory of your the two files:** python setup.py py2exe
6. Open the 'dist' directory and run your executable

A example setup.py script:

```
# Created 8-09-2011
# Ricky Savjani
# (savjani at bcm.edu)

#import necessary packages
from distutils.core import setup
import os, matplotlib
import py2exe

#the name of your .exe file
progName = 'MultipleSchizophrenia.py'

#Initialize Holder Files
preference_files = []
app_files = []
my_data_files=matplotlib.get_py2exe_datafiles()

#define which files you want to copy for data_files
```

(continues on next page)

(continued from previous page)

```

for files in os.listdir('C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-1.
↳65.00-py2.6.egg\\psychoy\\preferences\\'):
    f1 = 'C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-1.65.00-py2.6.
↳egg\\psychoy\\preferences\\' + files
    preference_files.append(f1)

#if you might need to import the app files
#for files in os.listdir('C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-
↳1.65.00-py2.6.egg\\psychoy\\app\\'):
#    f1 = 'C:\\Program Files\\PsychoPy2\\Lib\\site-packages\\PsychoPy-1.65.00-py2.6.
↳egg\\psychoy\\app\\' + files
#    app_files.append(f1)

#all_files = [("psychoy\\preferences", preference_files), ("psychoy\\app", app_
↳files), my_data_files[0]]

#combine the files
all_files = [("psychoy\\preferences", preference_files), my_data_files[0]]

#define the setup
setup(
    console=[progName],
    data_files = all_files,
    options = {
        "py2exe":{
            "skip_archive": True,
            "optimize": 2
        }
    }
)

```

13.6 Builder - providing feedback

If you're using the Builder then the way to provide feedback is with a *Code Component* to generate an appropriate message (and then a text to present that message). PsychoPy will be keeping track of various aspects of the stimuli and responses for you throughout the experiment and the key is knowing where to find those.

The following examples assume you have a *Loop* called *trials*, containing a *Routine* with a *Keyboard Component* called *key_resp*. Obviously these need to be adapted in the code below to fit your experiment.

Note: The following generate strings use python 'formatted strings'. These are very powerful and flexible but a little strange when you aren't used to them (they contain odd characters like `%2f`). See *Generating formatted strings* for more info.

13.6.1 Feedback after a trial

This is actually demonstrated in the demo, *ExtendedStroop* (in the Builder>demos menu, unpack the demos and then look in the menu again. tada!)

If you have a Keyboard Component called *key_resp* then, after every trial you will have the following variables:

```
key_resp.keys #a python list of keys pressed
key_resp.rt #the time to the first key press
key_resp.corr #None, 0 or 1, if you are using 'store correct'
```

To create your *msg*, insert the following into the ‘start experiment’ section of the *Code Component*:

```
msg='doh!'#if this comes up we forgot to update the msg!
```

and then insert the following into the *Begin Routine* section (this will get run every repeat of the routine):

```
if not key_resp.keys :
    msg="Failed to respond"
elif resp.corr:#stored on last run routine
    msg="Correct! RT=%.3f" %(resp.rt)
else:
    msg="Oops! That was wrong"
```

13.6.2 Feedback after a block

In this case the feedback routine would need to come after the loop (the block of trials) and the message needs to use the stored data from the loop rather than the *key_resp* directly. Accessing the data from a loop is not well documented but totally possible.

In this case, to get all the keys pressed in a *numpy* array:

```
trials.data['key_resp.keys'] #numpy array with size=[ntrials,ntypes]
```

If you used the ‘Store Correct’ feature of the Keyboard Component (and told psychopy what the correct answer was) you will also have a variable:

```
#numpy array storing whether each response was correct (1) or not (0)
trials.data['resp.corr']
```

So, to create your *msg*, insert the following into the ‘start experiment’ section of the *Code Component*:

```
msg='doh!'#if this comes up we forgot to update the msg!
```

and then insert the following into the *Begin Routine* section (this will get run every repeat of the routine):

```
nCorr = trials.data['key_resp.corr'].sum() #.std(), .mean() also available
meanRt = trials.data['key_resp.rt'].mean()
msg = "You got %i trials correct (rt=%.2f)" %(nCorr,meanRt)
```

13.6.3 Draw your message to the screen

Using one of the above methods to generate your *msg* in a *Code Component*, you then need to present it to the participant by adding a text to your *feedback* Routine and setting its text to *\$msg*.

Warning: The Text Component needs to be below the Code Component in the Routine (because it needs to be updated after the code has been run) and it needs to *set every repeat*.

13.7 Builder - terminating a loop

People often want to terminate their *Loops* before they reach the designated number of trials based on subjects' responses. For example, you might want to use a Loop to repeat a sequence of images that you want to continue until a key is pressed, or use it to continue a training period, until a criterion performance is reached.

To do this you need a *Code Component* inserted into your *routine*. All loops have an attribute called *finished* which is set to *True* or *False* (in Python these are really just other names for *1* and *0*). This *finished* property gets checked on each pass through the loop. So the key piece of code to end a loop called *trials* is simply:

```
trials.finished=True #or trials.finished=1 if you prefer
```

Of course you need to check the condition for that with some form of *if* statement.

Example 1: You have a change-blindness study in which a pair of images flashes on and off, with intervening blanks, in a loop called *presentationLoop*. You record the key press of the subject with a *Keyboard Component* called *resp1*. Using the 'ForceEndTrial' parameter of *resp1* you can end the current cycle of the loop but to end the loop itself you would need a *Code Component*. Insert the following two lines in the *End Routine* parameter for the Code Component, which will test whether more than zero keys have been pressed:

```
if resp1.keys is not None and len(resp1.keys)>0 :
    trials.finished=1
```

or:

```
if resp1.keys :
    presentationLoop.finished=1
```

Example 2: Sometimes you may have more possible trials than you can actually display. By default, a loop will present all possible trials (*nReps* * *length-of-list*). If you only want to present the first 10 of all possible trials, you can use a code component to count how many have been shown, and then finish the loop after doing 10.

This example assumes that your loop is named 'trials'. You need to add two things, the first to initialize the count, and the second to update and check it.

Begin Experiment:

```
myCount = 0
```

Begin Routine:

```
myCount = myCount + 1
if myCount > 10:
    trials.finished = True
```

Note: In Python there is no *end* to finish an *if* statement. The content of the *if* or of a for-loop is determined by the indentation of the lines. In the above example only one line was indented so that one line will be executed if the statement evaluates to *True*.

13.8 Installing PsychoPy in a classroom (administrators)

For running PsychoPy in a classroom environment it is probably preferable to have a ‘partial’ network installation. The PsychoPy library features frequent new releases, including bug fixes and you want to be able to update machines with these new releases. But PsychoPy depends on many other python libraries (over 200Mb in total) that tend not to change so rapidly, or at least not in ways critical to the running of experiments. If you install the whole PsychoPy application on the network then all of this data has to pass backwards and forwards, and starting the app will take even longer than normal.

The basic aim of this document is to get to a state whereby:

- Python and the major dependencies of PsychoPy are installed on the local machine (probably a disk image to be copied across your lab computers)
- PsychoPy itself (only ~2Mb) is installed in a network location where it can be updated easily by the administrator
- a file is created in the installation that provides the path to the network drive location
- Start-Menu shortcuts need to be set to point to the local Python but the remote PsychoPy application launcher

Once this is done, the vast majority of updates can be performed simply by replacing the PsychoPy library on the network drive.

13.8.1 1. Install dependencies locally

Download the latest version of the Standalone PsychoPy distribution, and run as administrator. This will install a copy of Python and many dependencies to a default location of *C:\Program Files\PsychoPy2*

13.8.2 2. Move the PsychoPy to the network

You need a network location that is going to be available, with read-only access, to all users on your machines. You will find all the contents of PsychoPy itself at something like this (version dependent obviously): *C:\Program Files\PsychoPy2\Lib\site-packages\PsychoPy-1.70.00-py2.6.egg*

Move that entire folder to your network location and call it *psychopyLib* (or similar, getting rid of the version-specific part of the name). Now the following should be a valid path: *<NETWORK_LOC>\psychopyLib\psychopy*

13.8.3 3. Update the Python path

The Python installation (in *C:\Program Files\PsychoPy2*) needs to know about the network location. If Python finds a text file with extension *.pth* anywhere on its existing path then it will add to the path any valid paths it finds in the file. So create a text file that has one line in it: *<NETWORK_LOC>\psychopyLib*

You can test if this has worked. Go to *C:\Program Files\PsychoPy2* and double-click on *python.exe*. You should get a Python terminal window come up. Now try:

```
>>> import psychopy
```

If psychopy is not found on the path then there will be an import error. Try adjusting the .pth file, restarting python.exe and importing again.

13.8.4 4. Update the Start Menu

The shortcut in the Windows Start Menu will still be pointing to the local (now non-existent) PsychoPy library. Right-click it to change properties and set the shortcut to point to something like:

```
"C:\Program Files\PsychoPy2\pythonw.exe" "<NETWORK_LOC>\psychopyLib\psychopy\app\  
↳psychopyApp.py"
```

You probably spotted from this that the PsychoPy app is simply a Python script. You may want to update the file associations too, so that .psyexp and .py are opened with:

```
"C:\Program Files\PsychoPy2\pythonw.exe" "<NETWORK_LOC>\psychopyLib\psychopy\app\  
↳psychopyApp.py" "%1"
```

Lastly, to make the shortcut look pretty, you might want to update the icon too. Set the icon's location to:

```
"<NETWORK_LOC>\psychopyLib\psychopy\app\Resources\psychopy.ico"
```

13.8.5 5. Updating to a new version

Fetch the latest .zip release. Unpack it and replace the contents of <NETWORK_LOC>\psychopyLib\ with the contents of the zip file.

13.9 Generating formatted strings

A formatted string is a variable which has been converted into a string (text). In python the specifics of how this is done is determined by what kind of variable you want to print.

Example 1: You have an experiment which generates a string variable called *text*. You want to insert this variable into a string so you can print it. This would be achieved with the following code:

```
message = 'The result is %s' %(text)
```

This will produce a variable *message* which if used in a text object would print the phrase 'The result is' followed by the variable *text*. In this instance %s is used as the variable being entered is a string. This is a marker which tells the script where the variable should be entered. %text tells the script which variable should be entered there.

Multiple formatted strings (of potentially different types) can be entered into one string object:

```
longMessage = 'Well done %s that took %0.3f seconds' %(info['name'], time)
```

Some of the handy formatted string types:

```
>>> x=5  
>>> x1=5124  
>>> z='someText'  
>>> 'show %s' %(z)  
'show someText'  
>>> '%0.1f' %(x) #will show as a float to one decimal place  
'5.0'
```

(continues on next page)

(continued from previous page)

```
>>> '%3i' %(x) #an integer, at least 3 chars wide, padded with spaces
'  5'
>>> '%03i' %(x) #as above but pad with zeros (good for participant numbers)
'005'
```

See the [python documentation](#) for a more complete list.

13.10 Coder - interleave staircases

Often psychophysicists using staircase procedures want to interleave multiple staircases, either with different start points, or for different conditions.

There is now a class, `psychopy.data.MultiStairHandler` to allow simple access to interleaved staircases of either basic or QUEST types. That can also be used from the *Loops* in the *Builder*. The following method allows the same to be created in your own code, for greater options.

The method works by nesting a pair of loops, one to loop through the number of trials and another to loop across the staircases. The staircases can be shuffled between trials, so that they do not simply alternate.

Note: Note the need to create a *copy* of the info. If you simply do `thisInfo=info` then all your staircases will end up pointing to the same object, and when you change the info in the final one, you will be changing it for all.

```
from psychopy import visual, core, data, event
from numpy.random import shuffle
import copy, time #from the std python libs

#create some info to store with the data
info={}
info['startPoints']=[1.5,3,6]
info['nTrials']=10
info['observer']='jwp'

win=visual.Window([400,400])
#-----
#create the stimuli
#-----

#create staircases
stairs=[]
for thisStart in info['startPoints']:
    #we need a COPY of the info for each staircase
    #(or the changes here will be made to all the other staircases)
    thisInfo = copy.copy(info)
    #now add any specific info for this staircase
    thisInfo['thisStart']=thisStart #we might want to keep track of this
    thisStair = data.StairHandler(startVal=thisStart,
        extraInfo=thisInfo,
        nTrials=50, nUp=1, nDown=3,
        minVal = 0.5, maxVal=8,
        stepSizes=[4,4,2,2,1,1])
    stairs.append(thisStair)

for trialN in range(info['nTrials']):
```

(continues on next page)

(continued from previous page)

```

    shuffle(stairs) #this shuffles 'in place' (ie stairs itself is changed, nothing_
↳returned)
    #then loop through our randomised order of staircases for this repeat
    for thisStair in stairs:
        thisIntensity = next(thisStair)
        print('start=%.2f, current=%.4f' %(thisStair.extraInfo['thisStart'],_
↳thisIntensity))

        #-----
        #run your trial and get an input
        #-----
        keys = event.waitKeys() #(we can simulate by pushing left for 'correct')
        if 'left' in keys: wasCorrect=True
        else: wasCorrect = False

        thisStair.addData(wasCorrect) #so that the staircase adjusts itself

    #this trial (of all staircases) has finished
#all trials finished

#save data (separate pickle and txt files for each staircase)
dateStr = time.strftime("%b_%d_%H%M", time.localtime())#add the current time
for thisStair in stairs:
    #create a filename based on the subject and start value
    filename = "%s start%.2f %s" %(thisStair.extraInfo['observer'], thisStair.
↳extraInfo['thisStart'], dateStr)
    thisStair.saveAsPickle(filename)
    thisStair.saveAsText(filename)

```

13.11 Making isoluminant stimuli

From the mailing list (see there for names, etc):

Q1: How can I create colours (RGB) that are isoluminant?

A1: The easiest way to create isoluminant stimuli (or control the luminance content) is to create the stimuli in DKL space and then convert them into RGB space for presentation on the monitor.

More details on DKL space can be found in the section about *Color spaces* and conversions between DKL and RGB can be found in the API reference for *psychopy.misc*

Q2: There's a difference in luminance between my stimuli. How could I correct for that?

I'm running an experiment where I manipulate color chromatic saturation, keeping luminance constant. I've coded the colors (red and blue) in rgb255 for 6 saturation values (10%, 20%, 30%, 40%, 50%, 60%, 90%) using a conversion from HSL to RGB color space.

Note that we don't possess spectrophotometers such as PR650 in our lab to calibrate each color gun. I've calibrated the gamma of my monitor psychophysically. Gamma was set to 1.7 (threshold) for `gamma(lum)`, `gamma(R)`, `gamma(G)`, `gamma(B)`. Then I've measured the luminance of each stimuli with a Brontes colorimeter. But there's a difference in luminance between my stimuli. How could I correct for that?

A2: Without a spectroradiometer you won't be able to use the color spaces like DKL which are designed to help this sort of thing.

If you don't care about using a specific colour space though you should be able to deduce a series of isoluminant colors manually, because the luminance outputs from each gun should sum linearly. e.g. on my monitor:


```
maxR=46cd/m2
maxG=114
maxB=15
```

(note that green is nearly always brightest)

So I could make a 15cd/m2 stimulus using various appropriate fractions of those max values (requires that the screen is genuinely gamma-corrected):

```
R=0, G=0, B=255
R=255*15/46, G=0, B=0
R=255*7.5/46, G=255*15/114, B=0
```

Note that, if you want a pure fully-saturated blue, then you're limited by the monitor to how bright you can make your stimulus. If you want brighter colours your blue will need to include some of the other guns (similarly for green if you want to go above the max luminance for that gun).

A2.1. You should also consider that even if you set appropriate RGB values to display your pairs of chromatic stimuli at the same luminance that they might still appear different, particularly between observers (and even if your light measurement device says the luminance is the same, and regardless of the colour space you want to work in). To make the pairs perceptually isoluminant, each observer should really determine their own isoluminant point. You can do this with the minimum motion technique or with heterochromatic flicker photometry.

13.12 Coder - show images

Sometimes we want to use PsychoPy to show images — either read from an image file (such as *.png*) or from a Numpy array. We can use either the *psychopy.visual.ImageStim* or *psychopy.visual.GratingStim* to achieve this. However, some of the nuances of actually getting the correct image to screen can be difficult to figure out.

This recipe demonstrates (1) a way to use *psychopy.visual.ImageStim* to read an image from disc and show it, and (2) using *psychopy.visual.ImageStim* to show a numpy array as an image.

When showing and converting images, you need to be careful about data types and channels. The [scikit-image docs on this](#) are quite good.

```
from pathlib import Path

import numpy as np
from PIL import Image
from psychopy import core, event, visual

# -----
# Setup window
# -----
win = visual.Window(
    (900, 900),
    screen=0,
    units="pix",
    allowGUI=True,
    fullscr=False,
)
# -----
# Example 1: load a stimulus from disk
# -----
```

(continues on next page)

(continued from previous page)

```

# assume we're running from the root psychopy repo.
# you could replace with path to any image:
path_to_image_file = Path() / "PsychoPy2_screenshot.png"

# simply pass the image path to ImageStim to load and display:
image_stim = visual.ImageStim(win, image=path_to_image_file)
text_stim = visual.TextStim(
    win,
    text="Showing image from file",
    pos=(0.0, 0.8),
    units="norm",
    height=0.05,
    wrapWidth=0.8,
)

image_stim.draw()
text_stim.draw()
win.flip()
event.waitKeys() # press space to continue

# -----
# Example 2: convert image to numpy array
#
# Perhaps you want to convert an image to numpy, do some things to it,
# and then display. Here I use the Python Imaging Library for image loading,
# and a conversion function from skimage. PsychoPy has an internal
# "image2array" function but this only handles single-layer (i.e. intensity) images.
#
# -----

pil_image = Image.open(path_to_image_file)
image_np = np.array(
    pil_image, order="C"
) # convert to numpy array with shape width, height, channels
image_np = (
    image_np.astype(np.float) / 255.0
) # convert to float in 0--1 range, assuming image is 8-bit uint.

# Note this float conversion is "quick and dirty" and will not
# fix potential out-of-range problems if you're going
# straight from a numpy array. See the img_as_float
# function of scikit image for a more careful conversion.

# flip image (row-axis upside down so we need to reverse it):
image_np = np.flip(image_np, axis=0)
image_stim = visual.ImageStim(
    win,
    image=image_np,
    units="pix",
    size=(
        image_np.shape[1],
        image_np.shape[0],
    ), # here's a gotcha: need to pass the size (x, y) explicitly.
    colorSpace="rgb1", # img_as_float converts to 0:1 range, whereas PsychoPy
    ↪ defaults to -1:1.
)
    
```

(continues on next page)

(continued from previous page)

```

text_stim.text = "Showing image from numpy array"
image_stim.draw()
text_stim.draw()
win.flip()
event.waitKeys() # press space to continue

win.close()
core.quit()

```

13.13 Adding a web-cam

From the mailing list (see there for names, etc):

“I spent some time today trying to get a webcam feed into my psychopy proj, inside my visual.window. The solution involved using the opencv module, capturing the image, converting that to PIL, and then feeding the PIL into a SimpleImageStim and looping and win.flipping. Also, to avoid looking like an Avatar in my case, you will have to change the default decoder used in PIL fromstring to utilize BGR instead of RGB in the decoding. I thought I would save some time for people in the future who might be interested in using a webcam feed for their psychopy project. All you need to do is import the opencv module into psychopy (importing modules was well documented by psychopy online) and integrate something like this into your psychopy script.”

```

from psychopy import visual, event
import Image, pylab, cv

mywin = visual.Window(allowGUI=False, monitor='testMonitor', units='norm',
                      colorSpace='rgb', color=[-1, -1, -1], fullscr=True)
mywin.setMouseVisible(False)

capture = cv.CaptureFromCAM(0)
img = cv.QueryFrame(capture)
pi = Image.fromstring(
    "RGB", cv.GetSize(img), img.tostring(), "raw", "BGR", 0, 1)
print(pi.size)
myStim = visual.GratingStim(win=mywin, tex=pi, pos=[0, 0.5], size=[0.6, 0.6],
                             opacity=1.0, units='norm')
myStim.setAutoDraw(True)

while True:
    img = cv.QueryFrame(capture)
    pi = Image.fromstring(
        "RGB", cv.GetSize(img), img.tostring(), "raw", "BGR", 0, 1)
    myStim.setTex(pi)
    mywin.flip()
    theKey = event.getKeys()
    if len(theKey) != 0:
        break

```


FREQUENTLY ASKED QUESTIONS (FAQS)

14.1 Why is the bits++ demo not working?

So far supports bits++ only in the bits++ mode (rather than mono++ or color++). In this mode, a code (the T-lock code) is written to the lookup table on the bits++ device by drawing a line at the top of the window. The most likely reason that the demo isn't working for you is that this line is not being detected by the device, and so the lookup table is not being modified. Most of these problems are actually nothing to do with /per se/, but to do with your graphics card and the CRS bits++ box itself.

There are a number of reasons why the T-lock code is not being recognised:

- the bits++ device is in the wrong mode. Open the utility that CRS supply and make sure you're in the right mode. Try resetting the bits++ (turn it off and on).
- the T-lock code is not fully on the screen. If you create a window that's too big for the screen or badly positioned then the code will be broken/not visible to the device.
- the T-lock code is on an 'odd' pixel.
- the graphics card is doing some additional filtering (win32). Make sure you turn off any filtering in the advanced display properties for your graphics card
- the gamma table of the graphics card is not set to be linear (but this should normally be handled by , so don't worry so much about it).
- you've got a Mac that's performing temporal dithering (new Macs, around 2009). Apple have come up with a new, very annoying idea, where they continuously vary the pixel values coming out of the graphics card every frame to create additional intermediate colours. This will break the T-lock code on 1/2-2/3rds of frames.

14.2 Can run my experiment with sub-millisecond timing?

This question is common enough and complex enough to have a section of the manual all of its own. See *Timing Issues and synchronisation*

14.3 How do I run experiments written with older versions of ?

If you perform experiments on computers shared by different research groups (e. g. at a shared experimental facility), it's possible that their experiments and yours are written using different versions of . Or maybe you yourself have some older and some newer experiments. In such a situation, it's important to ensure that the right version of is used for the right experiment.

In PsychoPy standalone, there is an easy-to-use system for controlling what version of is used.

14.3.1 Version control using Builder View

1. Open up the experiment file (a '.psyexp' file) that contains the experiment you want to use.
2. Go to experiment settings by clicking the icon with a cogwheel.
3. Under the "Basic" settings tab, there is an option named "Use PsychoPy version". Set it to the PsychoPy version you want to emulate.
4. Click "OK" to save the settings.
5. Run the experiment by clicking the green 'run' button.

14.3.2 Version control using Coder View

1. Open up the script file (a '.py' file) that contains the experiment you want to use.
2. Add `import psychopy` at the top of your script, **above** all other import statements.
3. Add the function call `psychopy.useVersion('<version_no>')` directly below `import psychopy`, but still **above** the other import statements. Here's an example:

```
import psychopy
psychopy.useVersion('2021.1.0')
from psychopy import visual, core, event
# the rest of your script follows
```

4. Run the script.

14.3.3 NOTE: Internet connection needed (the first time)

You need to have a working internet connection the first time that you run an experiment using a particular version (e. g. 1.90.2) on a computer, so that can download some info about the version for you. Once you've used a version once, your computer has saved the information it needs for emulating it. This means that after the first time, you don't need an internet connection if you run the same or another experiment using that version (e. g. 1.90.2).

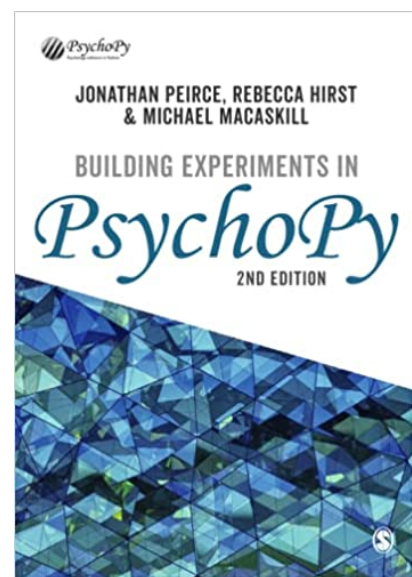
14.3.4 Compatibility

Using either of the above methods, you should often only need the latest version of standalone to run older experiments. However, if you have an experiment designed with a very old version of (say, version 1.77.01) you might have to install an older version of standalone . Since these things change over time, you probably want to search for help in the [PsychoPy forums](#).

RESOURCES (E.G. FOR TEACHING)

There are a number of further resources to help learn/teach about PsychoPy.

If you also have PsychoPy materials/course then please let us know so that we can link to them from here too!



15.1 Workshops

We are currently running virtual workshops in several formats, see our [workshops pages](#) for details

15.2 Youtube tutorials

- There is our [YouTube PsychoPy playlist](#) showing how to build basic experiments in the *Builder* interface.
- Jason Ozubko has added a series of great [PsychoPy Builder video tutorials](#) too
- Damien Mannion added a similarly great series of [PsychoPy programming videos](#) on YouTube
- ... and a [searching YouTube for PsychoPy](#) reveals many more!

15.3 Materials for Builder

The most comprehensive guide is the book [Building Experiments in PsychoPy](#) by Peirce, Hirst, and MacAskill.

The book is suitable for a wide range of needs and skill sets, with 3 sections for:

- The Beginner (suitable for undergraduate teaching)
- The Professional (more detail for creating more precise studies)
- The Specialist (with info about specialist needs such as studies in fMRI, EEG, ...)

At [School of Psychology, University of Nottingham](#), PsychoPy is now used for all first year practical class teaching. The classes that comprise that first year course are provided below. They were created partially with funding from the former [Higher Education Academy Psychology Network](#). Note that the materials here will be updated frequently as they are further developed (e.g. to update screenshots etc) so make sure you have the latest version of them!

- [PsychoPy_pracs_2011v2.zip](#) (21MB) (last updated: 15 Dec 2011)

The [GestaltReVision group \(University of Leuven\)](#) wiki covering PsychoPy (some Builder info and great tutorials for Python/PsychoPy coding of experiments).

There's a set of tools for teaching psychophysics using PsychoPy and a [PsychoPysics poster from VSS](#). Thanks James Ferwerda

15.4 Materials for Coder

- Please see the page on [officialWorkshops](#) for further details on coming to an intensive residential Python workshop in Nottingham.
- Marco Bertamini's book, [Programming Illusions for Everyone](#) is a fun way to learn about stimulus rendering in PsychoPy by learning how to create visual illusions
- [Gary Lupyan](#) runs a class on programming experiments using Python/PsychoPy and makes his lecture materials available [on this wiki](#)
- The [GestaltReVision group \(University of Leuven\)](#) offers a three-day crash course to Python and PsychoPy on a [IPython Notebook](#), and has lots of great information taking you from basic programming to advanced techniques.
- [Radboud University, Nijmegen](#) also has a [PsychoPy programming course](#)
- [Programming for Psychology in Python - Vision Science](#) has lessons and screencasts on PsychoPy (by Damien Mannion, UNSW Australia).

15.5 Previous events

- ECEM, August 2013 : Python for eye-tracking workshop with (Sol Simpson, Michael MacAskill and Jon Peirce). [Download Python-for-eye-tracking materials](#)
- VSS

For developers:

FOR DEVELOPERS

The best place to discuss ideas in depth is probably the dedicated [developers section of the forum](#)

For developers the best way to use is to install a version to your own copy of python (preferably 3.6 but we try to support a reasonable range). Make sure you have all the dependencies, including the extra suggested packages for developers.

Don't *install*. Instead fetch a copy of the git repository and add this to the python path using a .pth file. Other users of the computer might have their own standalone versions installed without your repository version touching them.

16.1 Using the repository

Any code that you want to be included into is done via Git in the GitHub repository. There's something of a learning curve to this, but it's common to development in many other packages.

For developers experienced with Git from other projects the only things you need to note are:

- the *PsychoPy Git Flow* for branches (which does not have a *master* or *main* but two branches, *dev* and *release*). **Please use the `release` branch as the base for bug fixes and the `dev` branch for feature development.**
- the format of *PsychoPy commit messages* is important so that we can see what changes have been made from a quick view of the git log

If your copy of the repository comes from before we used the 2-trunk GitFlow then you may also want to read the *Converting to the 2-trunk flow* section to update your repository to the new structure.

If you're new to git and/or contributing to open-source projects then you may want to go through as below:

- start at *Setting up your repository first time*
- *fixBugs*
- *Working on a new feature*
- *Making a pull request*

16.1.1 PsychoPy Git Flow

Unlike many projects, the PsychoPy repository has TWO main branches, *dev* and *release* (since Feb 2021). The design is similar to the [GitFlow workflow](#) except that we do not have anything named *master* (that is effectively now called *release*).

The system is designed to support our release pattern, with “feature releases” 2 or 3 times per year and bug-fix releases several times for each feature release. Major changes to the code, that potentially include new bugs, should not be included in bug-fix releases. So then the two main branches are as follows.

The dev branch: is for work that is going **to be held back for the next feature release**. Only fix bugs here if they are related to other un-released code or if the fixes require substantial code changes that might introduce new bugs. Those larger fixes will probably be held back for the next feature release. Simple bug fixes that get based on the *dev* branch might be hard to reincorporate back into the *release* branch .

The release branch: is for fixes that need **to be included in the next release**. It includes code changes that do not knowingly break/change existing experiments, and are small enough that we can be relatively confident that they do not introduce new bugs. Do not use this trunk for substantial pieces of development where new bugs might be introduced.

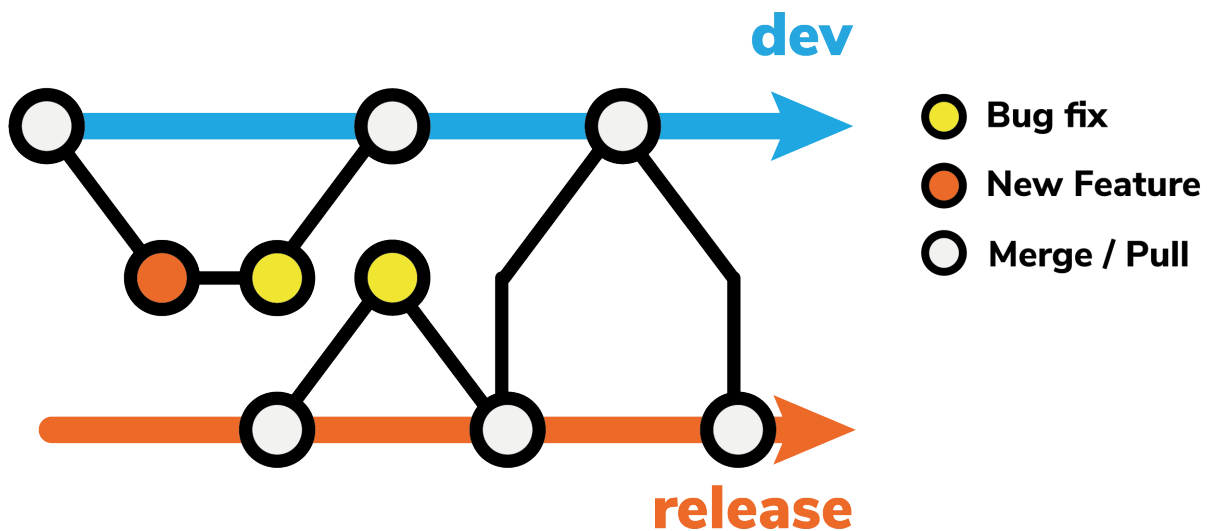


Fig. 16.1: Git Flow used by the PsychoPy project, with 2 main trunks for ‘dev’ and ‘release’. Bug fixes should be based on the *release* branch while new features or substantial code changes are built on the *dev* branch

Always create a branch for the work you are doing and take that branch from the tip of either *dev* or *release*.

Around a major (feature) release the two trunks will generally become synchronised.

16.1.2 PsychoPy commit messages

Informative commit messages are really useful when we have to go back through the repository finding the time that a particular change to the code occurred. Precede your message with one or more of the following:

- *BF* : bug fix
- *FF* : ‘feature’ fix. This is for fixes to code that hasn’t been released
- *RF* : refactoring
- *NF* : new feature
- *ENH* : enhancement (improvement to existing code)
- *DOC*: for all kinds of documentation-related commits
- *TEST*: for adding or changing tests

When making commits that fall into several commit categories (e.g., BF and TEST), **please make separate commits for each category** and **avoid concatenating commit message prefixes**. E.g., please do not use *BF/TEST*, because this will affect how commit messages are sorted when we pull in fixes for each release.

NB: The difference between BF and FF is that BF indicates a fix that is appropriate for back-porting to earlier versions, whereas FF indicates a fix to code that has not been released, and so cannot be back-ported.

So, a good commit message looks something like this. Note a) the commit title tells us what was fixed, the message tells us how that was achieved and includes a link to the GitHub issue if possible.

```
BF: fixed the updating of the stimulus position when units='deg'

The problem turned out to be that we had a typo in the attribute name

fixes GH-12323 [causes that GitHub issue to be closed and links them]
```

16.1.3 Setting up your repository first time

When you first start using the repo there are a few additional steps that you won’t need to do afterwards.

Create your own fork of the central repository

Go to [github](#), create an account and make a fork of the [psychopy repository](#) You can change your fork in any way you choose without it affecting the central project. You can also share your fork with others, including the central project.

Fetch a local copy

Install [git](#) on your computer. Create and upload an ssh key to your github account - this is necessary for you to push changes back to your fork of the project at github.

Then, in a folder of your choosing fetch your fork:

```
$ git clone git@github.com:USER/psychopy.git
$ cd psychopy
$ git remote add upstream git://github.com/psychopy/psychopy.git
```

The last line connects your copy (with read access) to the central server so you can easily fetch any updates to the central repository.

Run using your local repo copy

Now that you've fetched the latest version of psychopy using git, you should run this version in order to try out yours/others latest improvements. To use your github version all the time you should install that as a "developer" install so that the files stay in this location and as they get updated that is reflected in the installed version. This differs from a standard install where the files get copied to Python's site-packages and then changes you make have no effect until you install again. To run the developer install choose one of:

```
python -m pip install -e .      # to include the dependencies
python -m pip install -e --no-deps .  # to skip installing the dependencies
```

Run git version for just one session (Linux and Mac only): If you want to switch between a standard install and a development version from git you can choose to only temporarily run the git version. Open a terminal and set a temporary python path to your psychopy git folder:

```
$ export PYTHONPATH=/path/to/local/git/folder/
```

To check that worked you should open python in the terminal and try to import psychopy and see if it's the version you expected:

```
$ python
Python 3.6.8 (v3.6.8:3c6b436a57, Dec 24 2018, 02:04:31)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import psychopy
>>> print(psychopy.__version__)
2021.1.0
>>>
```

16.1.4 Fixing bugs and making minor improvements

To fix a bug in the main code, checkout the *release* trunk, create and checkout a new branch, then commit and push to your repo:

```
$ git checkout release
$ git checkout -b hotfix-whatAreYouFixing
  <do coding here and commits here>
$ git push origin release
```

Remember to use good commitMessage for your changes.

16.1.5 Working on a new feature

All substantial changes should be made on their own branch, coming from the *dev* trunk. Don't mix quick fixes with substantial changes with quick fixes (or with substantial changes on another topic). All changes should have their own branch so that we can then pick which ones we want to include and when.

To create a new branch:

```
$ git checkout dev # start from the tip of the dev trunk
$ git pull upstream dev # make sure we're up to date before we start
$ git checkout -b feature-somethingNew # create and checkout our new branch
<do coding here and commits here>
$ git push origin feature-somethingNew
```

Remember to use good commitMessage for your changes.

Once you've folded your new code back into your master and pushed it back to your github fork then it's time to *Making a pull request*.

16.1.6 Making a pull request

Once you've pushed your branch to your repository you can make a pull request from GitHub. If you go to your GitHub page for the repo it should be presenting you with a message explaining that there is new activity on the branch you just pushed, and that you might want to create a Pull Request. It's fairly simple form there. The rules about good commit messages don't even really apply to the Pull request itself, because it can be changed later more easily.

16.1.7 Converting to the 2-trunk flow

If you have an older copy of the repository with a *master* branch then you will need to follow these steps to get back in sync with the new *PsychoPy Git Flow*:. If you don't yet have a fork then don't worry - just go to *Setting up your repository first time*.

1. Update your fork on GitHub (if you haven't done that already): Visit <https://github.com/<yourUsername>/psychopy/branches> and select the pen next to *master* to rename it as *release*
2. Update your local branches to match the remote *release* trunk:

```
git branch -m master release # rename your local master to be release
git fetch origin # fetch the branches from your own remote
git branch -u origin/release release # set your renamed release to track origin/
↳release
```

- 3a. EITHER If you don't have a *dev* branch on your origin fork (i.e. first time you switch):

```
git fetch upstream # to get the dev branch from there
git checkout -b dev --track upstream/dev # create and checkout local dev from
↳upstream
git push -u origin dev
```

- 3b. OR If you already have a *dev* branch on your personal fork (e.g. you've converted another machine already):

```
git fetch origin # to get the dev branch from origin
git checkout -b dev --track origin/dev # create and checkout local dev from upstream
```

16.2 Adding documentation

There are several ways to add documentation, all of them useful: doc strings, comments in the code, and demos to show an example of actual usage. To further explain something to end-users, you can create or edit a *.rst* file that will automatically become formatted for the web, and eventually appear on www.psychopy.org.

You make a new file under *psychopy/docs/source/*, either as a new file or folder or within an existing one.

To test that your doc source code (*.rst* file) does what you expect in terms of formatting for display on the web, you can simply do something like (this is my actual path, unlikely to be yours):

```
$ cd /Users/jgray/code/psychopy/docs/
$ make html
```

Do this within your docs directory (requires sphinx to be installed, try “pip install sphinx” if it’s not working). That will add a build/html sub-directory.

Then you can view your new doc in a browser, e.g., for me:

`file:///Users/jgray/code/psychopy/docs/build/html/`

Push your changes to your github repository (using a “DOC:” commit message) and let Jon know, e.g. with a pull request.

16.3 Adding a new Builder Component

Builder Components are auto-detected and displayed to the experimenter as icons (in the right-most panel of the Builder interface panel). This makes it straightforward to add new ones.

All you need to do is create a list of parameters that the Component needs to know about (that will automatically appear in the Component’s dialog) and a few pieces of code specifying what code should be called at different points in the script (e.g. beginning of the Routine, every frame, end of the study etc. . .). Many of these will come simply from subclassing the `_base` or `_visual` Components.

To get started, *Working on a new feature* for the development of this component. (If this doesn’t mean anything to you then see *Using the repository*)

You’ll mainly be working in the directory `.../psychopy/experiment/components/`. Take a look at several existing Components (such as `image.py`), and key files including `_base.py` and `_visual.py`.

There are three main steps, the first being by far the most involved.

16.3.1 1. Create the file defining the component: `newcomp.py`

It’s most straightforward to model a new Component on one of the existing ones. Be prepared to specify what your Component needs to do at several different points in time: the first trial, every frame, at the end of each routine, and at the end of the experiment. In addition, you may need to sacrifice some complexity in order to keep things streamlined enough for a Builder (see e.g., `ratingscale.py`).

Your new Component class (in your file `newcomp.py`) should inherit from `BaseComponent` (in `_base.py`), `VisualComponent` (in `_visual.py`), or `KeyboardComponent` (in `keyboard.py`). You may need to rewrite some or all some of these methods, to override default behavior:

```
class NewcompComponent(BaseComponent): # or (VisualComponent)
    def __init__(...):
        super(NewcompComponent, self).__init__(...)
        ...
    def writeInitCode(self, buff):
    def writeRoutineStartCode(self, buff):
    def writeFrameCode(self, buff):
    def writeRoutineEndCode(self, buff):
```

Calling `super()` will create the basic default set of *params* that almost every component will need: `name`, `startVal`, `startType`, etc. Some of these fields may need to be overridden (e.g., `durationEstim` in `sound.py`). Inheriting from `VisualComponent` (which in turn inherits from `BaseComponent`) adds default visual params, plus arranges for Builder scripts to import `psychopy.visual`. If your component will need other libs, call `self.exp.requirePsychopyLib(['neededLib'])` (see e.g., `parallelPort.py`).

At the top of a component file is a dict named `_localized`. It contains mappings that allow a strict separation of internal string values (= used in logic, never displayed) from values used for display in the Builder interface (= for display only, possibly translated, never used in logic). The `.hint` and `.label` fields of `params['someParam']` should always be set to

a localized value, either by using a dict entry such as `_localized['message']`, or via the globally available translation function, `_(‘message’)`. Localized values must **not** be used elsewhere in a component definition.

Very occasionally, you may also need to edit `settings.py`, which writes out the set-up code for the whole experiment (e.g., to define the window). For example, this was necessary for the ApertureComponent, to pass `allowStencil=True` to the window creation.

Your new Component writes code into a buffer that becomes an executable python file, `xxx_lastrun.py` (where `xxx` is whatever the experimenter specifies when saving from the Builder, `xxx.psyexp`). You will do a bunch of this kind of call in your `newcomp.py` file:

```
buff.writeIndented(your_python_syntax_string_here)
```

You have to manage the indentation level of the output code, see `experiment.IndentingBuffer()`.

`xxx_lastrun.py` is the file that gets built when you run `xxx.psyexp` from the Builder. So you will want to look at `xxx_lastrun.py` frequently when developing your component.

Name-space

There are several internal variables (i.e. names of Python objects) that have a specific, hardcoded meaning within `xxx_lastrun.py`. You can expect the following to be there, and they should only be used in the original way (or something will break for the end-user, likely in a mysterious way):

```
win    # the window
t      # time within the trial loop, referenced to `trialClock`
x, y  # mouse coordinates, but only if the experimenter uses a mouse component
```

Handling of variable names is under active development, so this list may well be out of date. (If so, you might consider updating it or posting a note to the Discourse developer forum.)

Preliminary testing suggests that there are 600-ish names from numpy or numpy.random, plus the following:

```
['KeyResponse', '__builtins__', '__doc__', '__file__', '__name__', '__package__',
→ 'buttons', 'core', 'data', 'dlg', 'event', 'expInfo', 'expName', 'filename', 'gui',
→ 'logFile', 'os', 'psychopy', 'sound', 't', 'visual', 'win', 'x', 'y']
```

Yet other names get derived from user-entered names, like `trials` → `thisTrial`.

Params

`self.params` is a key construct that you build up in `__init__`. You need name, startTime, duration, and several other params to be defined or you get errors. `name` should be of type `code`.

The `Param()` class is defined in `psychopy.app.builder.experiment.Param()`. A very useful thing that Params know is how to create a string suitable for writing into the .py script. In particular, the `__str__` representation of a Param will format its value (`.val`) based on its type (`.valType`) appropriately. This means that you don't need to check or handle whether the user entered a plain string, a string with a code trigger character (`$`), or the field was of type `code` in the first place. If you simply request the `str()` representation of the param, it is formatted correctly.

To indicate that a param (eg, `thisParam`) should be considered as an advanced feature, set its category to advanced: `self.params['thisParam'].categ = 'Advanced'`. Then the GUI shown to the experimenter will automatically place it on the 'Advanced' tab. Other categories work similarly (`Custom`, etc).

During development, it can sometimes be helpful to save the params into the `xxx_lastrun.py` file as comments, so you can see what is happening:

```
def writeInitCode(self, buff):
    # for debugging during Component development:
    buff.writeIndented("# self.params for aperture:\n")
```

(continues on next page)

(continued from previous page)

```

for p in self.params:
    try: buff.writeIndented("# %s: %s <type %s>\n" % (p, self.params[p].val, self.
↪params[p].valType))
    except: pass

```

A lot more detail can be inferred from existing components.

Making things loop-compatible looks interesting – see *keyboard.py* for an example, especially code for saving data at the end.

16.3.2 Notes & gotchas

syntax errors in new_comp.py: The app will fail to start if there are syntax error in any of the components that are auto-detected. Just correct them and start the app again.

param[.val]: If you have a boolean variable (e.g., *my_flag*) as one of your params, note that *self.param["my_flag"]* is always True (the param exists → True). So in a boolean context you almost always want the *.val* part, e.g., *if self.param["my_flag"].val:*

However, you do not always want *.val*. Specifically, in a string/unicode context (= to trigger the self-formatting features of Param(s), you almost always want “%s” % *self.param["my_flag"]*, without *.val*. Note that it’s better to do this via “%s” than *str()* because *str(self.param["my_flag"])* coerces things to type str (squashing unicode) whereas %s works for both str and unicode.

Travis testing Before submitting a pull request with the new component, you should regenerate the *componsTemplate.txt* file. This is a text file that lists the attributes of all of the user interface settings and options in the various components. It is used during the Travis automated testing process when a pull request is submitted to GitHub, allowing the detection of errors that may have been caused in refactoring. Your new component needs to have entries added to this file if the Travis testing is going to pass successfully.

To re-generate the file, cd to this directory *.../psychopy/tests/test_app/test_builder/* and run:

```
`python genComponsTemplate.py --out`
```

This will over-write the existing file so you might want to make a copy in case the process fails.
Compatibility issues: As at May 2018, that script is not yet Python 3 compatible, and on a Mac you might need to use *pythonw*.

16.3.3 2. Icon: newcomp.png

Using your favorite image software, make an icon for your Component with a descriptive name, e.g., *newcomp.png*. Dimensions = 48 × 48. Put it in the components directory.

In *newcomp.py*, have a line near the top:

```
iconFile = path.join(thisFolder, 'newcomp.png')
```

16.3.4 3. Documentation: newcomp.rst

Just make a descriptively-named text file that ends in `.rst` (“restructured text”), and put it in `psychopy/docs/source/builder/components/`. It will get auto-formatted and end up at <https://www.psychopy.org/builder/components/newcomp.html>

16.4 Style-guide for coder demos

Each coder demo is intended to illustrate a key feature (or two), especially in ways that show usage in practice, and go beyond the description in the API. The aim is not to illustrate every aspect, but to get people up to speed quickly, so they understand how basic usage works, and could then play around with advanced features.

As a newcomer to , you are in a great position to judge whether the comments and documentation are clear enough or not. If something is not clear, you may need to ask a contributor for a description; email psychopy-dev@googlegroups.com.

Here are some style guidelines, written for the OpenHatch event(s) but hopefully useful after that too. These are intended specifically for the coder demos, not for the internal code-base (although they are generally quite close).

The idea is to have clean code that looks and works the same way across demos, while leaving the functioning mostly untouched. Some small changes to function might be needed (e.g., to enable the use of ‘escape’ to quit), but typically only minor changes like this.

- Generally, when you run the demo, does it look good and help you understand the feature? Where might there be room for improvement? You can either leave notes in the code in a comment, or include them in a commit message.
- Standardize the top stuff to have 1) a shebang with python, 2) utf-8 encoding, and 3) a comment:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""Demo name, purpose, description (1-2 sentences, although some demos need more
↳explanation).
"""
```

For the comment / description, it’s a good idea to read and be informed by the relevant parts of the API (see <https://psychopy.org/api/api.html>), but there’s no need to duplicate that text in your comment. If you are unsure, please post to the dev list psychopy-dev@googlegroups.com.

- Follow PEP-8 mostly, some exceptions:
 - current convention is to use camelCase for variable names, so don’t convert those to underscores
 - 80 char columns can spill over a little. Try to keep things within 80 chars most of the time.
 - do allow multiple imports on one line if they are thematically related (e.g., `import os, sys, glob`).
 - inline comments are ok (because the code demos are intended to illustrate and explain usage in some detail, more so than typical code).
- Check all imports:
 - remove any unnecessary ones
 - replace `import time` with `from psychopy import core`. Use `core.getTime()` (= ms since the script started) or `core.getAbsTime()` (= seconds, unix-style) instead of `time.time()`, for all time-related functions or methods not just `time()`.

- add `from __future__ import division`, even if not needed. And make sure that doing so does not break the demo!
- Fix any typos in comments; convert any lingering British spellings to US, e.g., change *colour* to *color*
- Prefer `if <boolean>`: as a construct instead of `if <boolean> == True:`. (There might not be any to change).
- If you have to choose, opt for more verbose but easier-to-understand code instead of clever or terse formulations. This is for readability, especially for people new to python. If you are unsure, please add a note to your commit message, or post a question to the dev list psychopy-dev@googlegroups.com.
- Standardize variable names:
 - use *win* for the `visual.Window()`, and so `win.flip()`
- Provide a consistent way for a user to exit a demo using the keyboard, ideally enable this on every visual frame: use `if len(event.getKeys(['escape'])): core.quit()`. **Note:** if there is a previous `event.getKeys()` call, it can slurp up the 'escape' keys. So check for 'escape' first.
- Time-out after 10 seconds, if there's no user response and a timeout is appropriate for the demo (and a longer time-out might be needed, e.g., for `ratingScale.py`):

```
demoClock = core.clock() # is demoClock's time is 0.000s at this point
...
if demoClock.getTime() > 10.:
    core.quit()
```

- Most demos are not full screen. For any that are full-screen, see if it can work without being full screen. If it has to be full-screen, add some text to say that pressing 'escape' will quit.
- If displaying log messages to the console seems to help understand the demo, here's how to do it:

```
from psychopy import logging
...
logging.console.setLevel(logging.INFO) # or logging.DEBUG for even more stuff
```

- End a script with `win.close()` (assuming the script used a `visual.Window`), and then `core.quit()` even though it's not strictly necessary

16.5 Localization (I18N, translation)

is used worldwide. Starting with v1.81, many parts of itself (the app) can be translated into any language that has a unicode character set. A translation affects what the experimenter sees while creating and running experiments; it is completely separate from what is shown to the subject. Translations of the online documentation will need a completely different approach.

In the app, translation is handled by a function, `_translate()`, which takes a string argument. (The standard name is `_()`, but unfortunately this conflicts with `_` as used in some external packages that depends on.) The `_translate()` function returns a translated, unicode version of the string in the locale / language that was selected when starting the app. If no translation is available for that locale, the original string is returned (= English).

A locale setting (e.g., 'ja_JP' for Japanese) allows the end-user (= the experimenter) to control the language that will be used for display within the app itself. (It can potentially control other display conventions as well, not just the language.) will obtain the locale from the user preference (if set), or the OS.

Workflow: 1) Make a translation from English (en_US) to another language. You'll need a strong understanding of , English, and the other language. 2) In some cases it will be necessary to adjust 's code, but only if new code has been added to the app and that code displays text. Then re-do step 1 to translate the newly added strings.

See notes in `psychopy/app/localization/readme.txt`.

16.5.1 Make a translation (.po file)

As a translator, you will likely introduce many new people to , and your translations will greatly influence their experience. Try to be completely accurate; it is better to leave something in English if you are unsure how is supposed to work.

To translate a given language, you'll need to know the standard 5-character code (see *psychopy/app/localization/mappings*). E.g., for Japanese, wherever LANG appears in the documentation here, you should use the actual code, i.e., "ja_JP" (without quotes).

A free app called poedit is useful for managing a translation. For a given language, the translation mappings (from en_US to LANG) are stored in a .po file (a text file with extension .po); after editing with poedit, these are converted into binary format (with extension .mo) which are used when the app is running.

- Start translation (do these steps once):

Start a translation by opening *psychopy/app/locale/LANG/LC_MESSAGE/messages.po* in Poedit. If there is no such .po file, create a new one:

- make a new directory *psychopy/app/locale/LANG/LC_MESSAGE/* if needed. Your LANG will be auto-detected within only if you follow this convention. You can copy metadata (such as the project name) from another .po file.

Set your name and e-mail address from "Preferences..." of "File" menu. Set translation properties (such as project name, language and charset) from Catalog Properties Dialog, which can be opened from "Properties..." of "Catalog" menu.

In poedit's properties dialog, set the "source keywords" to include '_translate'. This allows poedit to find the strings in that are to be translated.

To add paths where Poedit scans .py files, open "Sources paths" tab on the Catalog Properties Dialog, and set "Base path:" to "./././././" (= psychopy/psychopy/). Nothing more should be needed. If you've created new catalog, save your catalog to *psychopy/app/locale/LANG/LC_MESSAGE/messages.po*.

Probably not needed, but check anyway: Edit the file containing language code and name mappings, *psychopy/app/localization/mappings*, and fill in the name for your language. Give a name that should be familiar to people who read that language (i.e., use the name of the language as written in the language itself, not in en_US). About 25 are already done.

- Edit a translation:

Open the .po file with Poedit and press "Update" button on the toolbar to update newly added / removed strings that need to be translated. Select a string you want to translate and input your translation to "Translation:" box. If you are unsure where string is used, point on the string in "Source text" box and right-click. You can see where the string is defined.

- Technical terms should not be translated: Builder, Coder, , Flow, Routine, and so on. (See the Japanese translation for guidance.)
- If there are formatting arguments in the original string (`%s, %(first) i`), the same number of arguments must also appear in the translation (but their order is not constrained to be the original order). If they are named (e.g., `%(first) i`), that part should not be translated—here `first` is a python name.
- If you think your translation might have room for improvement, indicate that it is "fuzzy". (Saving Notes does not work for me on Mac, seems like a bug in poedit.)
- After making a new translation, saving it in poedit will save the .po file and also make an associated .mo file. You need to update the .mo file if you want to see your changes reflected in .

- The start-up tips are stored in separate files, and are not translated by poedit. Instead:
- copy the default version (named *psychopy/app/Resources/tips.txt*) to a new file in the same directory, named *tips_LANG.txt*. Then replace English-language tips with translated tips. Note that some of the humor might not translate well, so feel free to leave out things that would be too odd, or include occasional mild humor that would be more appropriate. Humor must be respectful and suitable for using in a classroom, laboratory, or other professional situation. Don't get too creative here. If you have any doubt, best leave it out. (Hopefully it goes without saying that you should avoid any religious, political, disrespectful, or sexist material.)
- in poedit, translate the file name: translate "tips.txt" as "tips_LANG.txt"
- Commit both the .po and .mo files to github (not just one or the other), and any changed files (e.g., *tips_LANG*, *localization/mappings*).

16.5.2 Adjust PsychoPy's code

This is mostly complete (as of 1.81.00), but will be needed for new code that displays text to users of the app (experimenters, not study participants).

There are a few things to keep in mind when working on the app's code to make it compatible with translations. If you are only making a translation, you can skip this section.

- In PsychoPy's code, the language to be used should always be English with American spellings (e.g., "color").
- Within the app, the return value from `_translate()` should be used only for display purposes: in menus, tooltips, etc. A translated value should never be used as part of the logic or internal functioning of . It is purely a "skin". Internally, everything must be in `en_US`.
- Basic usage is exactly what you expect: `_translate("hello")` will return a unicode string at run-time, using mappings for the current locale as provided by a translator in a .mo file. (Not all translations are available yet, see above to start a new one.) To have the app display a translated string to the experimenter, just display the return value from the underscore translation function.
- The strings to be translated must appear somewhere in the app code base as explicit strings within `_translate()`. If you need to translate a variable, e.g., named `str_var` using the expression `_translate(str_var)`, somewhere else you need to explicitly give all the possible values that `str_var` can take, and enclose each of them within the translate function. It is okay for that to be elsewhere, even in another file, but not in a comment. This allows poedit to discover of all the strings that need to be translated. (This is one of the purposes of the `_localized` dict at the top of some modules.)
- `_translate()` should not be given a null string to translate; if you use a variable, check that it is not '' to avoid invoking `_translate('')`.
- Strings that contain formatting placeholders (e.g., `%d`, `%s`, `%.4f`) require a little more thought. Single placeholders are easy enough: `_translate("hello, %s") % name`.
- Strings with multiple formatting placeholders require named arguments, because positional arguments are not always sufficient to disambiguate things depending on the phrase and the language to be translated into: `_translate("hello, %(first)s %(last)s") % {'first': firstname, 'last': lastname}`
- Localizing drop-down menus is a little more involved. Such menus should display localized strings, but return selected values as integers (`GetSelection()` returns the position within the list). Do not use `GetStringSelection()`, because this will return the localized string, breaking the rule about a strict separation of display and logic. See `Builder ParamDialogs` for examples.

16.5.3 Other notes

When there are more translations (and if they make the app download large) we might want to manage things differently (e.g., have translations as a separate download from the app).

16.6 Adding a new Menu Item

Adding a new menu-item to the Builder (or Coder) is relatively straightforward, but there are several files that need to be changed in specific ways.

16.6.1 1. makeMenus()

The code that constructs the menus for the Builder is within a method named *makeMenus()*, within class *builder.BuilderFrame()*. Decide which submenu your new command fits under, and look for that section (e.g., File, Edit, View, and so on). For example, to add an item for making the Routine panel items larger, I added two lines within the View menu, by editing the *makeMenus()* method of class *BuilderFrame* within *psychopy/app/builder/builder.py* (similar for Coder):

```
self.viewMenu.Append(self.IDs.tbIncrRoutineSize, _("&Routine Larger\t%s") %self.app.
↳keys['largerRoutine'], _("Larger routine items"))
wx.EVT_MENU(self, self.IDs.tbIncrRoutineSize, self.routinePanel.increaseSize)
```

Note the use of the translation function, *_()*, for translating text that will be displayed to users (menu listing, hint).

16.6.2 2. wxIDs.py

A new item needs to have a (numeric) ID so that *wx* can keep track of it. Here, the number is *self.IDs.tbIncrRoutineSize*, which I had to define within the file *psychopy/app/wxIDs.py*:

```
tbIncrRoutineSize=180
```

It's possible that, instead of hard-coding it like this, it's better to make a call to *wx.NewIdRef()* – *wx* will take care of avoiding duplicate IDs, presumably.

16.6.3 3. Key-binding prefs

I also defined a key to use to as a keyboard short-cut for activating the new menu item:

```
self.app.keys['largerRoutine']
```

The actual key is defined in a preference file. Because *psychopy* is multi-platform, you need to add info to four different *.spec* files, all of them being within the *psychopy/preferences/* directory, for four operating systems (Darwin, FreeBSD, Linux, Windows). For *Darwin.spec* (meaning macOS), I added two lines. The first line is not merely a comment: it is also automatically used as a tooltip (in the preferences dialog, under key-bindings), and the second being the actual short-cut key to use:

```
# increase display size of Routines
largerRoutine = string(default='Ctrl++') # on Mac Book Pro this is good
```

This means that the user has to hold down the *Ctrl* key and then press the + key. Note that on Macs, ‘Ctrl’ in the spec is automatically converted into ‘Cmd’ for the actual key to use; in the .spec, you should always specify things in terms of ‘Ctrl’ (and not ‘Cmd’). The default value is the key-binding to use unless the user defines another one in her or his preferences (which then overrides the default). Try to pick a sensible key for each operating system, and update all four .spec files.

16.6.4 4. Your new method

The second line within *makeMenus()* adds the key-binding definition into wx’s internal space, so that when the key is pressed, wx knows what to do. In the example, it will call the method *self.routinePanel.increaseSize*, which I had to define to do the desired behavior when the method is called (in this case, increment an internal variable and redraw the routine panel at the new larger size).

16.6.5 5. Documentation

To let people know that your new feature exists, add a note about your new feature in the CHANGELOG.txt, and appropriate documentation in .rst files.

16.7 Creating Plugins for

Plugins provide a means for developers to extend , adding new features and customizations without directly modifying the installation. Read usingplugins for more information about plugins before proceeding on this page.

16.7.1 How plugins work

The plugin system in functions as a dynamic importer, which imports additional executable code from plugin packages then patches them into an active session at runtime. This is done by calling the `psychopy.plugins.loadPlugin()` function and passing the project name of the desired plugin to it. Once `loadPlugin()` returns, imported objects are immediately accessible. Any changes made to with plugins do not persist across sessions, meaning if Python is restarted, will return to its default behaviour unless `loadPlugin()` is called again.

Installed plugins for are discoverable on the system using package metadata. The metadata of the package defines “entry points” which tells the plugin loader where within PsychoPy’s namespace to place objects exported by the plugin. The loader also ensures plugins are compatible with the Python environment (ie. operating system, CPU architecture, and Python version). Any Python package can define entry points, allowing developers to add functionality to without needing to create a separate plugin project.

16.7.2 Plugin packages

A plugin has a similar structure to Python package, see the official *Packaging Python Projects* (<https://packaging.python.org/tutorials/packaging-projects>) guide for details.

Naming plugin packages

Standalone plugins, which are packages that exist only to extend should adhere to the following naming convention to make plugins discernible from any other package in public repositories. Plugin project names should always be prefixed with *psychopy* with individual words separated with a - or _ symbol (i.e. *psychopy-quest-procedure* or *psychopy_quest_procedure* are valid). What you chose to name the package is up to you, but keep it concise and informative.

Note: The plugin system does not use project names to identify plugins, rather relying on package metadata to identify if a package has entry points pertinent to . Therefore, projects do not need to be named a particular way to still be used as plugins. This allows packages which are not primarily used with to extend it, without the need for a separate plugin package. It also allows a single package to be used as a plugin for multiple projects unrelated to .

The module or sub-package which defines the objects which entry points refer to should be some variant of the name to prevent possible namespace conflicts. For instance, we would name our module *psychopy_quest_procedure* if our project was called *psychopy-quest-procedure*.

Specifying entry points

Entry points reference objects in a plugin module that will attach to itself. Packages advertise their entry points by having them in their metadata. How entry points are defined and added to package metadata is described in the section [Dynamic Discovery of Services and Plugins](#) of the documentation for *setuptools*.

When loading a specified plugin, the plugin loader searches for a distribution matching the given project name, then gets the entry point mapping from its metadata. Any entry point belonging to groups whose names start with *psychopy* is loaded. Group names are fully-qualified names of modules or unbound classes within PsychoPy's namespace to create links to the associated entry points in the plugin module/package.

As an example, using entry point groups and specifiers, we can add a class called *MyStim* defined in the plugin module *psychopy_plugin* to appear in *psychopy.visual* when the plugin is loaded. To do this, we use the following dictionary when defining entry point metadata with the *setup()* function in the plugin project's *setup.py* file:

```
setup(
    ...
    entry_points={'psychopy.visual': 'MyStim = psychopy_plugin:MyStim'},
    ...
)
```

Note: Plugins can load and assign entry points to names anywhere in PsychoPy's namespace. However, plugin developers should place them where they make most sense. In the last example, we put *MyStim* in *psychopy.visual* because that's where users would expect to find it if it was part of the base installation.

If we have additional classes we'd like to add to *psychopy.visual*, entry entry points for that group can be given as a list of specifiers:

```
setup(
    ...
    entry_points={
        'psychopy.visual': ['MyStim = psychopy_plugin:MyStim',
                           'MyStim2 = psychopy_plugin:MyStim2']
    },
    ...
)
```

For more complex (albeit contrived) example to demonstrate how to modify unbound class attributes (ie. methods and properties), say we have a plugin which provides a custom interface to some display hardware called *psychopy-display* that needs to alter the existing `flip()` method of the `psychopy.visual.Window` class to work. Furthermore, we want to add a class to *psychopy.hardware* called *DisplayControl* to give the user a way of setting up and configuring the display. Entry points for both objects are defined in the plugin's *psychopy_display* module. To get the effect we want, we specify entry points using the following:

```
setup(
    ...
    entry_points={
        'psychopy.visual.Window': ['flip = psychopy_display:flip'],
        'psychopy.hardware': ['DisplayControl = psychopy_display:DisplayControl']},
    ...
)
```

After calling `loadPlugin('psychopy-display')`, the user will be able to create instances of `psychopy.hardware.DisplayControl` and new instances of `psychopy.visual.Window` will have the modified `flip()` method.

The `__register__` attribute

Plugin modules can define an optional attribute named `__register__` which specifies a callable object. The purpose of `__register__` is to allow the module to perform tasks before loading entry points based on arguments passed to it by the plugin loader. The arguments passed to the target of `__register__`, come from the `**kwargs` given to `loadPlugins()`. The value of this attribute can be a string of the name or a reference to a callable object (ie. function or method).

Note: The `__register__` attribute should only ever be used for running routines pertinent to setting up entry points. The referenced object is only called on a module once per session.

As an example, consider a case where an entry point is defined as `doThis` in plugin *python-foobar*. There are two possible behaviors which are *foo* and *bar* that `dothis` can have. We can implement both behaviors in separate functions, and use arguments passed to the `__register__` target to assign which to use to as the entry point:

```
__register__ = 'register'

doThis = None

def foo():
    return 'foo'

def bar():
    return 'bar'

def register(**kwargs):
    global dothis
    option = kwargs.get('option', 'foo')
    if option == 'bar':
        dothis = bar
    else:
        dothis = foo
```

When the user calls `loadPlugin('python-foobar', option='bar')`, the plugin will assign function `bar()` to `doThis`. If `option` is not specified or given as 'foo', the behavior of `doThis` will be that of `foo()`.

16.7.3 Plugin example project

This section will demonstrate how to create a plugin project and package it for distribution. For this example, we will create a plugin called *psychopy-rect-area* which adds a method to the `psychopy.visual.Rect` stimulus class called *getArea()* that returns the area of the shape when called.

Project files

First, we need to create a directory called *psychopy-rect-area* which all our Python packages and code will reside. Inside that directory, we create the following files and directories:

```
psychopy-rect-area/
  psychopy_rect_area/
    __init__.py
  MANIFEST.in
  README.md
  setup.py
```

The implementation for the *getArea()* method will be defined in a file called `psychopy_rect_area/__init__.py`, it should contain the following:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Plugin entry points for `psychopy-rect-area`."""

def get_area(self):
    """Compute the area of a `Rect` stimulus in `units`.

    Returns
    -----
    float
        Area in units^2.

    """
    return self.size[0] * self.size[1]
```

Note: The *get_area()* function needs to have *self* as the first argument because we are going to assign it as class method. All class methods get a reference to the class as the first argument. You can name this whatever you like (eg. *cls*).

The `setup.py` script is used to generate an installable plugin package. This should contain something like the following:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from setuptools import setup

setup(name='psychopy-rect-area',
      version='1.0',
      description='Compute the area of a Rect stimulus.',
      long_description='',
      url='http://repo.example.com',
      author='Nobody',
      author_email='nobody@example.com',
```

(continues on next page)

(continued from previous page)

```
license='GPL3',
classifiers=[
    'Development Status :: 4 - Beta',
    'License :: OSI Approved :: GLP3 License',
    'Programming Language :: Python :: 2.7',
    'Programming Language :: Python :: 3'
],
keywords='psychopy stimulus',
packages=['psychopy_rect_area'],
install_requires=['psychopy'],
include_package_data=True,
entry_points={
    'psychopy.visual.Rect': ['getArea = psychopy_rect_area:get_area']
},
zip_safe=False)
```

Looking at `entry_points` we can see that we are assigning `psychopy_rect_area.get_area` to `psychopy.visual.Rect.getArea`. Attributes assigned to entry points should follow the naming conventions of (camel case), however plugins are free to use internally whatever style the author chooses (eg. PEP8). You should also use appropriate classifiers for your plugin, a full list can be found here (https://pypi.org/pypi/?%3Aaction=list_classifiers).

You can also specify `install_requires` to indicate which versions of PsychPy are compatible with your plugin. Visit <https://packaging.python.org/discussions/install-requires-vs-requirements/> for more information.

One should also include a `README.md` file which provides detailed information about the plugin. This file can be read and passed to the `long_description` argument of `setup()` in `setup.py` if desired by inserting the following into the setup script:

```
from setuptools import setup

def get_readme_text():
    with open('README.md') as f:
        return f.read()

setup(
    ...
    long_description=get_readme_text(),
    ...
)
```

Finally, we need specify `README.md` in our `MANIFEST.in` file to tell the packaging system to include the file when packaging. Simply put the following line in `MANIFEST.in`:

```
README.md
```

Building packages

plugin packages are built like any other Python package. We can build a *wheel* distribution by calling the following console command:

```
python setup.py sdist bdist_wheel
```

The resulting `.whl` files will appear in directory `psychopy-rect-area/dist`. The generated packages can be installed with `pip` or uploaded to the [Python Package Index](https://pypi.org/). for more information about building and uploading packages, visit: <https://packaging.python.org/tutorials/packaging-projects/>

If uploaded to PyPI, other users can install your plugin by entering the following into their command prompt:

```
python -m pip install psychopy-rect-area
```

Using the plugin

Once installed the plugin can be activated by using the `psychopy.plugins.loadPlugin()` function. This function should be called after the import statements in your script:

```
from psychopy import visual, core, plugins
plugins.loadPlugin('psychopy-demo-plugin') # load the plugin
```

After calling `loadPlugin()`, all instances of `Rect` will have the method `getArea()`:

```
rectStim = visual.Rect(win)
rectArea = rectStim.getArea()
```

16.7.4 Plugins as patches

A special use case of plugins is to apply and distribute “patches”. Using entry points to override module and class attributes, one can create patches to fix minor bugs in extant installations between releases, or backport fixes and features to older releases (that support plugins) that cannot be upgraded for some reason. Patches can be distributed like any other Python package, and can be installed and applied uniformly across multiple installations.

Plugins can also patch other plugins that have been previously loaded by `loadPlugin()` calls. This is done by defining entry points to module and class attributes that have been created by a previously loaded plugin.

Creating patches

As an example, consider a fictional scenario where a bug was introduced in a recent release of by a hardware vendor updating their drivers. As a result, PsychoPy’s builtin support for their devices provided by the `psychopy.hardware.Widget` class is now broken. You notice that it has been fixed in a pending release of , and that it involves a single change to the `getData()` method of the `psychopy.hardware.Widget` class to get it working exactly as before. However, you cannot wait for the next release because you are in the middle of running scheduled experiments, even worse, you have dozens of test stations using the hardware.

In this case, you can create a plugin to not only fix the bug, but apply it across multiple existing installations to save the day. Creating a package for our patch is no different than a regular plugin (see the *Plugin example project* section for more information), so you go about creating a project for a plugin called *psychopy-hotfix* which defines the working version of the `getData()` method in a sub-module called `psychopy_hotfix` like this:

```
# method copy and pasted from the bug fix commit
def getData(self):
    """This function reads data from the device."""
    # code here ...
```

In the `setup.py` file of the plugin package, specify the entry points like this to override the defective method in our installations:

```
setup(
    name='psychopy-hotfix'
    ...
    entry_points={
```

(continues on next page)

(continued from previous page)

```

        'psychopy.hardware.Widget': ['getData = psychopy_patch:getData']
    },
    ...
)

```

That's it, just build a distributable package and install it on all the systems affected by the bug.

Applying patches

Whether you create your own patch, or obtain one provided by the community, they are applied using the `loadPlugin()` function after installing them. Experiment scripts will need to have the following lines added under the `import` statements at the top of the file for the plugin to take effect (but it's considerably less work than manually patching in the code across many separate installations):

```

import psychopy.plugin as plugin
plugin.loadPlugin('psychopy-patch')

```

After `loadPlugin` is called, the behaviour of the `getData()` method of any instances of the `psychopy.hardware.Widget` class will change to the correct one.

Once a new release of comes out with the patch incorporated into it and your installations are upgraded, you can remove the above lines.

16.7.5 Creating window backends

Custom backends for the `Window` class can be implemented in plugins, allowing one to create windows using frameworks other than Pyglet, GLFW, and PyGame that can be enabled using the appropriate `winType` argument.

A plugin can add a `winType` by specifying class and module entry points for `psychopy.visual.backends`. If the entry point is a subclass of `psychopy.visual.backends.BaseBackend` and has `winTypeName` defined, it will be automatically registered and can be used as a `winType` by instances of `psychopy.visual.Window`.

Note: If a module is given as an entry point, the whole module will be added to `backends` and any class within it that is a subclass of `BaseBackend` and defines `winTypeName` will be registered. This allows one to add multiple window backends to with a single plugin module.

Example

For example, say we have a backend class called `CustomBackend` defined in module `custom_backend` in the plugin package `psychopy-custom-backend`. We can tell the plugin loader to register it to be used when a `Window` instance is created with `winType='custom'` by adding the `winTypeName` class attribute to `CustomBackend`:

```

class CustomBackend(BaseBackend):
    winTypeName = 'custom'
    ...

```

Note: If `winTypeName` is not defined, the entry points will still get added to `backends` but users will not be able to use it directly by specifying `winType`.

We define the entry point for our custom backend in `setup.py` as:

```

setup(
    ...
    entry_points={
        'psychoPy.visual.backends': 'custom_backend = custom_backend'},
    ...
)

```

Optionally, we can point to the backend class directly:

```

setup(
    ...
    entry_points={
        'psychoPy.visual.backends':
            'custom_backend = custom_backend:CustomBackend'},
    ...
)

```

After the plugin is installed and loaded, we can use our backend for creating windows by specifying `winType` as `winTypeName`:

```

loadPlugin('psychoPy-custom-backend')
win = Window(winType='custom')

```

16.8 Contributing to the Test Suite

16.8.1 Why do we need a test suite?

With any bit of software, no matter how perfect the code seems as you're writing it, there will be bugs. We use a test suite to make sure that we find as many of those bugs as we can before users do, it's always better to catch them in development than to have them mess up someone's experiment once the software is out in the wild. Remember - when a user finds a bug, they react like this:



Fig. 16.2: “Starship Troopers” (TriStar Pictures; Touchstone Pictures)

... but when the test suite finds a bug, developers react like this:



Fig. 16.3: “Birds In A Nest” (Robert Lynch)

The more bugs the test suite finds, the better!

16.8.2 How does it work?

The test suite uses a Python module called [pytest](https://pypi.org/project/pytest/) to run tests on various parts of the code. These tests work by calling functions, initialising objects and generally trying to use as much of the code in the PsychoPy repo as possible - then, if an uncaught error is hit at any point, *pytest* will spit out some informative text on what went wrong. This means that, if the test suite can run without error, then the software can do everything done in the test suite without error.

To mark something as a test, it needs three things:

1. It must be somewhere in the folder *psychopy/psychopy/tests*
2. It must contain the word *test* in its name (i.e. the class name and function names)
4. It must be executable in code, a function or a method

So, for example, if you were to make a test for the *visual.Rect* class, you might call the file *test_rect.py* and put it in *psychopy/psychopy/tests/test_all_visual*, and the file might look like this:

```
from psychopy import visual # used to draw stimuli

def test_rect():
    # Test that we can create a window and a rectangle without error
    win = visual.Window()
    rect = visual.Rect(win)
    # Check that they draw without error
    rect.draw()
    win.flip()
    # End test
    win.close()
```

16.8.3 Using assert

Sometimes there's more to a bit of code than just running without error - we need to check not just that it doesn't crash, but that the output is as expected. The *assert* function allows us to do this. Essentially, *assert* will throw an *AssertionError* if the first input is *False*, with the text of this error determined by the second input. So, for example:

```
assert 2 < 1, "2 is not less than 1"
```

will raise:

```
AssertionError: 2 is not less than 1
```

In essence, an *assert* call is the same as saying:

```
if condition == False:
    raise AssertionError(msg)
```

What this means is that we can raise an error if a value is not what we expect it to be, which will cause the test to fail if the output of a function is wrong, even if the function ran without error.

You could use *assert* within the *test_rect* example like so:

```
# Set the rectangle's fill color
rect.colorSpace = 'rgb'
rect.fillColor = (1, -1, -1)
# Check that the rgb value of its fill color is consistent with what we set
assert rect._fillColor == colors.Color('red'), f"Was expecting rect._fillColor to_
↪have an rgb value of '(1, -1, -1)', but instead it was '{rect._fillColor.rgb}'"
```


Meaning that, if something was wrong with *visual.Rect* such that setting its *fillColor* attribute didn't set the rgb value of its fill color correctly, this test would raise an *AssertionError* and would print both the expected and actual values. This process of comparing actual outputs against expected outputs is known as “end-to-end” (e2e) testing, while simply supplying values to see if they cause an error is called “unit” testing.

16.8.4 Using classes

In addition to individual methods, you can also create a *class* for tests. This approach is useful when you want to avoid making loads of objects for each test, as you can simply create an object once and then refer back to it. For example:

```
class TestRect:
    """ A class to test the Rect class """
    @classmethod
    def setup_class(self):
        """ Initialise the rectangle and window objects """
        # Create window
        self.win = visual.Window()
        # Create rect
        self.rect = visual.Rect(self.win)

    def test_color(self):
        """ Test that the color of a rectangle sets correctly """
        # Set the rectangle's fill color
        self.rect.colorSpace = 'rgb'
        self.rect.fillColor = (1, -1, -1)
        # Check that the rgb value of its fill color is consistent with what we set
        assert self.rect._fillColor == colors.Color('red'), f"Was expecting rect._
↪fillColor to have an rgb value of '(1, -1, -1)'," \
        f" but instead it was '{self.rect._
↪fillColor.rgb}'"
```

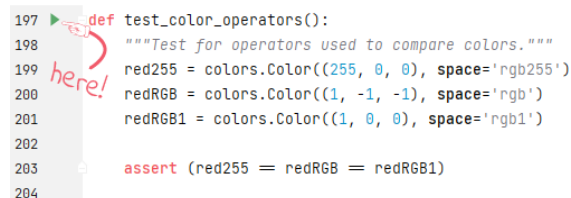
Of course, you could create a window and a rectangle for each function and it would work just the same, but only creating one means the test suite doesn't have as much to do so it will run faster. Test classes work the same as any other class definition, except that rather than `__init__`, the constructor function should be *setup_class*, and this should be marked as a *@classmethod* as in the example above.

Exercise

Practicing writing tests? Try extending the above class to test if a created rectangle has 4 vertices.

16.8.5 Running tests in PyCharm

One of the really useful features on PyCharm is its ability to run tests with just a click. If you have *pytest* installed, then any valid test will have a green play button next to its name, in the line margins:



```
197 ▶ def test_color_operators():
198     """Test for operators used to compare colors."""
199     red255 = colors.Color((255, 0, 0), space='rgb255')
200     redRGB = colors.Color((1, -1, -1), space='rgb')
201     redRGB1 = colors.Color((1, 0, 0), space='rgb1')
202
203     assert (red255 == redRGB == redRGB1)
204
```

Clicking this button will start all the necessary processes to run this test, just like it would run in our test suite. This button also appears next to test classes, clicking the run button next to the class name will create an instance of that class, then run each of its methods which are valid tests.

16.8.6 Test utils

The test suite comes with some handy functions and variables to make testing easier, all of which can be accessed by importing `psychopy.tests.utils`.

Paths

The test utils module includes the following paths:

- `TESTS_PATH` : A path to the root tests folder
- `TESTS_DATA_PATH` : A path to the data folder within the tests folder - here is where all screenshots, example conditions files, etc. for use by the test suite are stored

Compare screenshot

This function allows you to compare the appearance of a *visual.Window* to an image file, raising an *AssertionError* if they aren't sufficiently similar. This takes three arguments:

- `fileName` : A path to the image you want to compare against
- `win` : The window you want to check
- `crit` (optional) : A measure of how lenient to be - this defaults to 5, but we advise increasing it to 20 for anything involving fonts as these can vary between machines

If `filename` points to a file which doesn't exist, then this function will instead save the window and assume true. Additionally, if the comparison fails, the window will be saved as the same path as `filename`, but with `_local` appended to the name.

Compare pixel color

Sometimes, comparing an entire image may be excessive for what you want to check. For example, if you just want to make sure that a fill color has applied, you could just compare the color of one pixel. This means there doesn't need to be a `.png` file in the PsychoPy repository, and the test suite also doesn't have to load a entire image just to compare one color. In these instances, it's better to use `utils.comparePixelColor`. This function takes three arguments:

- `screen` : The window you want to check
- `color` : The color you expect the pixel to be (ideally, this should be a `colors.Color` object)
- `coord` (optional) : The coordinates of the pixel within the image which you're wanting to compare (defaults to `(0, 0)`)

Contained within this function is an `assert` call - so if the two colors are not the same, it will raise an *AssertionError* giving you information on both the target color and the pixel color.

Exemplars and tykes

While you're welcome to lay out your tests however makes the most sense for that test, a useful format in some cases it to define *list's of "exemplars" and "tykes" - 'dict's of attributes for use in a 'for loop*, to save yourself from manually writing the same code over and over, with "exemplars" being very typical use cases which should definitely work as a bare minimum, and "tykes" being edge cases which should work but are not necessarily likely to occur. Here's an example of this structure:

```

from psychopy import visual, colors # used to draw stimuli

class TestRect:
    """ A class to test the Rect class """
    @classmethod
    def setup_class(self):
        """ Initialise the rectangle and window objects """
        # Create window
        self.win = visual.Window()
        # Create rect
        self.rect = visual.Rect(self.win)

    def test_color(self):
        """ Test that the color or a rectangle sets correctly """
        # Set the rectangle's fill color
        self.rect.colorSpace = 'rgb'
        self.rect.fillColor = (1, -1, -1)
        # Check that the rgb value of its fill color is consistent with what we set
        assert self.rect._fillColor == colors.Color('red'), f"Was expecting rect._
↪fillColor to have an rgb value of '(1, -1, -1)'," \
        f" but instead it was '{self.rect._
↪fillColor.rgb}'"

    def test_rect_colors(self):
        """Test a range of known exemplar colors as well as colors we know to be
↪troublesome AKA tykes"""
        # Define exemplars
        exemplars = [
            { # Red with a blue outline
              'fill': 'red',
              'border': 'blue',
              'colorSpace': 'rgb',
              'targetFill': colors.Color((1, -1, -1), 'rgb'),
              'targetBorder': colors.Color((-1, -1, 1), 'rgb'),
            },
            { # Blue with a red outline
              'fill': 'blue',
              'border': 'red',
              'colorSpace': 'rgb',
              'targetFill': colors.Color((-1, -1, 1), 'rgb'),
              'targetBorder': colors.Color((1, -1, -1), 'rgb'),
            },
        ]
        # Define tykes
        tykes = [
            { # Transparent fill with a red border when color space is hsv
              'fill': None,
              'border': 'red',
            }
        ]
    
```

(continues on next page)

(continued from previous page)

```

        'colorSpace': 'rgb',
        'targetFill': colors.Color(None, 'rgb'),
        'targetBorder': colors.Color((0, 1, 1), 'hsv'),
    }
]
# Iterate through all exemplars and tykes
for case in exemplars + tykes:
    # Set colors
    self.rect.colorSpace = case['colorSpace']
    self.rect.fillColor = case['fill']
    self.rect.borderColor = case['border']
    # Check values are the same
    assert self.rect._fillColor == case['targetFill'], f"Was expecting rect._
↪fillColor to be '{case['targetFill']}', but instead it was '{self.rect._fillColor}'"
    assert self.rect._borderColor == case['targetBorder'], f"Was expecting_
↪rect._borderColor to be '{case['targetBorder']}', but instead it was '{self.rect._
↪borderColor}'"

```

16.8.7 Cleanup

After opening any windows, initialising objects or opening any part of the app, it's important to do some cleanup afterwards - otherwise these won't close and the test suite will just keep running forever. This just means calling `.Close()` on any `wx.Frame`'s, `.close()` on any `visual.Window`'s, and using `del` to get rid of any objects.

For functions, you can just do this at the end of the function, before it terminates. For classes, this needs to be done in a method called `teardown_class`; as `pytest` will call this method when the tests have completed. This method also needs to have a decorator marking it as a *classfunction*, like so:

```

from psychopy import visual

class ExampleTest:
    def __init__(self):
        # Start an app
        wx.App()
        # Create a frame
        self.frame = wx.Frame()
        # Create a window
        self.win = visual.Window()
        # Create an object
        self.rect = visual.Rect(win)

    @classmethod
    def teardown_class(self):
        # Close the frame
        self.frame.Close()
        # Close the window
        self.win.close()
        # Delete the object
        del self.rect

```

Exercise

Add a `teardown_class` method to your `TestRect` class.

16.8.8 CodeCov

CodeCov is a handy tool which runs the full test suite and keeps track of which lines of code are executed - giving each file in the PsychoPy repo a percentage score for “coverage”. If more lines of code in that file are executed when the test suite runs, then it has a higher coverage score. You can view the full coverage report for the repo [here](<https://app.codecov.io/gh/psychopy/psychopy/>).

Some areas of the code are more important than others, so it’s important not to make decisions purely based on what most increases coverage, but coverage can act as a good indicator for what areas the test suite is lacking in. If you want to make a test but aren’t sure what to do, finding a file or folder with a poor coverage score is a great place to start!

Solutions

Testing if a created rectangle has 4 vertices:

```
def test_rect(self):
    """ Test that a rect object has 4 vertices """
    assert len(self.rect.vertices) == 4, f"Was expecting 4 vertices in a Rect object,
↳ got {len(self.rect.vertices)}"
```

Adding a `teardown_class` method to your `TestRect` class:

```
class TestRect:
    """ A class to test the Rect class """
    @classmethod
    def setup_class(self):
        """ Initialise the rectangle and window objects """
        # Create window
        self.win = visual.Window()
        # Create rect
        self.rect = visual.Rect(self.win)

    def test_color(self):
        """ Test that the color of a rectangle sets correctly """
        # Set the rectangle's fill color
        self.rect.colorSpace = 'rgb'
        self.rect.fillColor = (1, -1, -1)
        # Check that the rgb value of its fill color is consistent with what we set
        assert self.rect._fillColor == colors.Color('red'), f"Was expecting rect._
↳ fillColor to have an rgb value of '(1, -1, -1)'," \
            f" but instead it was '{self.rect._
↳ fillColor.rgb}'"

    def test_rect(self):
        """ Test that a rect object has 4 vertices """
        assert len(self.rect.vertices) == 4, f"Was expecting 4 vertices in a Rect
↳ object, got {len(self.rect.vertices)}"

    def test_rect_colors(self):
        """Test a range of known exemplar colors as well as colors we know to be
↳ troublesome AKA tykes"""
```

(continues on next page)

```

# Define exemplars
exemplars = [
    { # Red with a blue outline
      'fill': 'red',
      'border': 'blue',
      'colorSpace': 'rgb',
      'targetFill': colors.Color((1, -1, -1), 'rgb'),
      'targetBorder': colors.Color((-1, -1, 1), 'rgb'),
    },
    { # Blue with a red outline
      'fill': 'blue',
      'border': 'red',
      'colorSpace': 'rgb',
      'targetFill': colors.Color((-1, -1, 1), 'rgb'),
      'targetBorder': colors.Color((1, -1, -1), 'rgb'),
    },
]

# Define tykes
tykes = [
    { # Transparent fill with a red border when color space is hsv
      'fill': None,
      'border': 'red',
      'colorSpace': 'rgb',
      'targetFill': colors.Color(None, 'rgb'),
      'targetBorder': colors.Color((0, 1, 1), 'hsv'),
    }
]

# Iterate through all exemplars and tykes
for case in exemplars + tykes:
    # Set colors
    self.rect.colorSpace = case['colorSpace']
    self.rect.fillColor = case['fill']
    self.rect.borderColor = case['border']
    # Check values are the same
    assert self.rect._fillColor == case['targetFill'], f"Was expecting rect._
↪fillColor to be '{case['targetFill']}', but instead it was '{self.rect._fillColor}'"
    assert self.rect._borderColor == case['targetBorder'], f"Was expecting_
↪rect._borderColor to be '{case['targetBorder']}', but instead it was '{self.rect._
↪borderColor}'"

    @classmethod
    def teardown_class(self):
        """clean-up any objects, wxframes or windows opened by the test"""
        # Close the window
        self.win.close()
        # Delete the object
        del self.rect

```

Happy Coding Folks!!

EXPERIMENT FILE FORMAT (.PSYEXP)

The file format used to save experiments constructed in builder was created especially for the purpose, but is an open format, using a basic xml form, that may be of use to other similar software. Indeed the builder itself could be used to generate experiments on different backends (such as Vision Egg, PsychToolbox or PyEPL). The xml format of the file makes it extremely platform independent, as well as moderately(?) easy to read by humans. There was a further suggestion to generate an XSD (or similar) [schema against which psyexp files could be validated](#). That is a low priority but welcome addition if you wanted to work on it(!) There is a basic XSD (XML Schema Definition) available in *psychopy/app/builder/experiment.xsd*.

The simplest way to understand the file format is probably simply to create an experiment, save it and open the file in an xml-aware editor/viewer (e.g. change the file extension from .psyexp to .xml and then open it in Firefox). An example (from the stroop demo) is shown below.

The file format maps fairly obviously onto the *structure of experiments* constructed with the *Builder* interface. There are general *Settings* for the experiment, then there is a list of *Routines* and a *Flow* that describes how these are combined.

As with any xml file the format contains object *nodes* which can have direct properties and also child nodes. For instance the outermost node of the .psyexp file is the experiment node, with properties that specify the version of that was used to save the file most recently and the encoding of text within the file (ascii, unicode etc.), and with child nodes *Settings*, *Routines* and *Flow*.

17.1 Parameters

Many of the nodes described within this xml description of the experiment contain Param entries, representing different parameters of that Component. Nearly all parameter nodes have a *name* property and a *val* property. The parameter node with the name “advancedParams” does not have them. Most also have a *valType* property, which can take values ‘bool’, ‘code’, ‘extendedCode’, ‘num’, ‘str’ and an *updates* property that specifies whether this parameter is changing during the experiment and, if so, whether it changes ‘every frame’ (of the monitor) or ‘every repeat’ (of the Routine).

17.2 Settings

The Settings node contains a number of parameters that, in , would normally be set in the *Experiment settings* dialog, such as the monitor to be used. This node contains a number of *Parameters* that map onto the entries in that dialog.

17.3 Routines

This node provides a sequence of xml child nodes, each of which describes a *Routine*. Each Routine contains a number of children, each specifying a *Component*, such as a stimulus or response collecting device. In the *Builder* view, the *Routines* obviously show up as different tabs in the main window and the *Components* show up as tracks within that tab.

17.4 Components

Each *Component* is represented in the .psyexp file as a set of parameters, corresponding to the entries in the appropriate component dialog box, that completely describe how and when the stimulus should be presented or how and when the input device should be read from. Different *Components* have slightly different nodes in the xml representation which give rise to different sets of parameters. For instance the *TextComponent* nodes has parameters such as *colour* and *font*, whereas the *KeyboardComponent* node has parameters such as *forceEndTrial* and *correctIf*.

17.5 Flow

The Flow node is rather more simple. Its children simply specify objects that occur in a particular order in time. A Routine described in this flow must exist in the list of Routines, since this is where it is fully described. One Routine can occur once, more than once or not at all in the Flow. The other children that can occur in a Flow are LoopInitiators and LoopTerminators which specify the start and endpoints of a loop. All loops must have exactly one initiator and one terminator.

17.6 Names

For the experiment to generate valid code the name parameters of all objects (Components, Loops and Routines) must be unique and contain no spaces. That is, an experiment can not have two different Routines called 'trial', nor even a Routine called 'trial' and a Loop called 'trial'.

The Parameter names belonging to each Component (or the Settings node) must be unique within that Component, but can be identical to parameters of other Components or can match the Component name themselves. A TextComponent should not, for example, have multiple 'pos' parameters, but other Components generally will, and a Routine called 'pos' would also be also permissible.

```
<PsychoPy2experiment version="1.50.04" encoding="utf-8">
  <Settings>
    <Param name="Monitor" val="testMonitor" valType="str" updates="None"/>
    <Param name="Window size (pixels)" val="[1024, 768]" valType="code" updates="None" />
    <Param name="Full-screen window" val="True" valType="bool" updates="None"/>
    <Param name="Save log file" val="True" valType="bool" updates="None"/>
    <Param name="Experiment info" val="{ 'participant': 's_001', 'session': 001 }"
    valType="code" updates="None"/>
    <Param name="Show info dlg" val="True" valType="bool" updates="None"/>
    <Param name="logging level" val="warning" valType="code" updates="None"/>
    <Param name="Units" val="norm" valType="str" updates="None"/>
    <Param name="Screen" val="1" valType="num" updates="None"/>
  </Settings>
  <Routines>
```

(continues on next page)

(continued from previous page)

```

<Routine name="trial">
  <TextComponent name="word">
    <Param name="name" val="word" valType="code" updates="constant"/>
    <Param name="text" val="thisTrial.text" valType="code" updates="set every_
↳repeat"/>
    <Param name="colour" val="thisTrial.rgb" valType="code" updates="set every_
↳repeat"/>
    <Param name="ori" val="0" valType="code" updates="constant"/>
    <Param name="pos" val="[0, 0]" valType="code" updates="constant"/>
    <Param name="times" val="[0.5,2.0]" valType="code" updates="constant"/>
    <Param name="letterHeight" val="0.2" valType="code" updates="constant"/>
    <Param name="colourSpace" val="rgb" valType="code" updates="constant"/>
    <Param name="units" val="window units" valType="str" updates="None"/>
    <Param name="font" val="Arial" valType="str" updates="constant"/>
  </TextComponent>
  <KeyboardComponent name="resp">
    <Param name="storeCorrect" val="True" valType="bool" updates="constant"/>
    <Param name="name" val="resp" valType="code" updates="None"/>
    <Param name="forceEndTrial" val="True" valType="bool" updates="constant"/>
    <Param name="times" val="[0.5,2.0]" valType="code" updates="constant"/>
    <Param name="allowedKeys" val="['1','2','3']" valType="code" updates="constant
↳"/>
    <Param name="storeResponseTime" val="True" valType="bool" updates="constant"/>
    <Param name="correctIf" val="resp.keys==str(thisTrial.corrAns)" valType="code
↳" updates="constant"/>
    <Param name="store" val="last key" valType="str" updates="constant"/>
  </KeyboardComponent>
</Routine>
<Routine name="instruct">
  <TextComponent name="instrText">
    <Param name="name" val="instrText" valType="code" updates="constant"/>
    <Param name="text" val="&quot;Please press;&#10;1 for red ink,&#10;2 for_
↳green ink&#10;3 for blue ink&#10;(Esc will quit)&#10;&#10;Any key to continue&quot;
↳" valType="code" updates="constant"/>
    <Param name="colour" val="[1, 1, 1]" valType="code" updates="constant"/>
    <Param name="ori" val="0" valType="code" updates="constant"/>
    <Param name="pos" val="[0, 0]" valType="code" updates="constant"/>
    <Param name="times" val="[0, 10000]" valType="code" updates="constant"/>
    <Param name="letterHeight" val="0.1" valType="code" updates="constant"/>
    <Param name="colourSpace" val="rgb" valType="code" updates="constant"/>
    <Param name="units" val="window units" valType="str" updates="None"/>
    <Param name="font" val="Arial" valType="str" updates="constant"/>
  </TextComponent>
  <KeyboardComponent name="ready">
    <Param name="storeCorrect" val="False" valType="bool" updates="constant"/>
    <Param name="name" val="ready" valType="code" updates="None"/>
    <Param name="forceEndTrial" val="True" valType="bool" updates="constant"/>
    <Param name="times" val="[0, 10000]" valType="code" updates="constant"/>
    <Param name="allowedKeys" val="" valType="code" updates="constant"/>
    <Param name="storeResponseTime" val="False" valType="bool" updates="constant"/
↳>
    <Param name="correctIf" val="resp.keys==str(thisTrial.corrAns)" valType="code
↳" updates="constant"/>
    <Param name="store" val="last key" valType="str" updates="constant"/>
  </KeyboardComponent>
</Routine>
<Routine name="thanks">

```

(continues on next page)

```

<TextComponent name="thanksText">
  <Param name="name" val="thanksText" valType="code" updates="constant"/>
  <Param name="text" val="&quot;Thanks!&quot;" valType="code" updates="constant
↪"/>

  <Param name="colour" val="[1, 1, 1]" valType="code" updates="constant"/>
  <Param name="ori" val="0" valType="code" updates="constant"/>
  <Param name="pos" val="[0, 0]" valType="code" updates="constant"/>
  <Param name="times" val="[1.0, 2.0]" valType="code" updates="constant"/>
  <Param name="letterHeight" val="0.2" valType="code" updates="constant"/>
  <Param name="colourSpace" val="rgb" valType="code" updates="constant"/>
  <Param name="units" val="window units" valType="str" updates="None"/>
  <Param name="font" val="arial" valType="str" updates="constant"/>
</TextComponent>
</Routine>
</Routines>
<Flow>
  <Routine name="instruct"/>
  <LoopInitiator loopType="TrialHandler" name="trials">
    <Param name="endPoints" val="[0, 1]" valType="num" updates="None"/>
    <Param name="name" val="trials" valType="code" updates="None"/>
    <Param name="loopType" val="random" valType="str" updates="None"/>
    <Param name="nReps" val="5" valType="num" updates="None"/>
    <Param name="trialList" val="['text': 'red', 'rgb': [1, -1, -1], 'congruent': ↵
↪1, 'corrAns': 1], {'text': 'red', 'rgb': [-1, 1, -1], 'congruent': 0, 'corrAns': 1},
↪ {'text': 'green', 'rgb': [-1, 1, -1], 'congruent': 1, 'corrAns': 2}, {'text':
↪ 'green', 'rgb': [-1, -1, 1], 'congruent': 0, 'corrAns': 2}, {'text': 'blue', 'rgb': ↵
↪ [-1, -1, 1], 'congruent': 1, 'corrAns': 3}, {'text': 'blue', 'rgb': [1, -1, -1],
↪ 'congruent': 0, 'corrAns': 3}]" valType="str" updates="None"/>
    <Param name="trialListFile" val="/Users/jwp...troop/trialTypes.csv" valType="str
↪" updates="None"/>
  </LoopInitiator>
  <Routine name="trial"/>
  <LoopTerminator name="trials"/>
  <Routine name="thanks"/>
</Flow>
</PsychoPy2experiment>

```

PYTHON MODULE INDEX

p

- `psychopy.clock`, 152
- `psychopy.core`, 149
- `psychopy.data`, 696
- `psychopy.hardware`, 464
 - `psychopy.hardware.brainproducts`, 467
 - `psychopy.hardware.crs`, 471
 - `psychopy.hardware.emulator`, 505
 - `psychopy.hardware.forp`, 510
 - `psychopy.hardware.joystick`, 511
 - `psychopy.hardware.keyboard`, 464
 - `psychopy.hardware.minolta`, 515
 - `psychopy.hardware.pr`, 517
 - `psychopy.hardware.qmix`, 521
- `psychopy.info`, 746
- `psychopy.iohub.client`, 523
 - `psychopy.iohub.client.keyboard`, 530
- `psychopy.logging`, 754
- `psychopy.misc`, 759
- `psychopy.parallel`, 767
- `psychopy.preferences`, 772
- `psychopy.sound`, 440
- `psychopy.tools`, 581
 - `psychopy.tools.colorspacetools`, 581
 - `psychopy.tools.coordinatetools`, 587
 - `psychopy.tools.filetools`, 587
 - `psychopy.tools.gltools`, 588
 - `psychopy.tools.imagetools`, 634
 - `psychopy.tools.mathtools`, 634
 - `psychopy.tools.monitorunittools`, 677
 - `psychopy.tools.plottools`, 678
 - `psychopy.tools.typetools`, 680
 - `psychopy.tools.unittools`, 680
 - `psychopy.tools.viewtools`, 682
- `psychopy.visual.filters`, 741
- `psychopy.visual.windowframepack`, 436
- `psychopy.visual.windowwarp`, 436

Symbols

- `_EOS()` (*psychopy.sound.backend_ptb.SoundPTB* method), 443
- `_EOS()` (*psychopy.sound.backend_sounddevice.SoundDeviceSound* method), 445
- `_Goggles()` (*psychopy.hardware.crs.bits.BitsPlusPlus* method), 472
- `_Goggles()` (*psychopy.hardware.crs.bits.BitsSharp* method), 482
- `_Logger` (class in *psychopy.logging*), 754
- `_RTBoxDecodeResponse()` (*psychopy.hardware.crs.bits.BitsSharp* method), 482
- `_ResetClock()` (*psychopy.hardware.crs.bits.BitsPlusPlus* method), 472
- `_ResetClock()` (*psychopy.hardware.crs.bits.BitsSharp* method), 483
- `__init__()` (*psychopy.tools.gltools.ObjMeshInfo* method), 622
- `__init__()` (*psychopy.tools.gltools.QueryObjectInfo* method), 598
- `__init__()` (*psychopy.tools.gltools.VertexArrayInfo* method), 609
- `__init__()` (*psychopy.tools.gltools.VertexBufferInfo* method), 612
- `_addDeviceView()` (*psychopy.iohub.client.ioHubConnection* method), 529
- `_assignFlipTime()` (*psychopy.visual.Window* method), 421
- `_assignFlipTime()` (*psychopy.visual.nnivs.VisualSystemHD* method), 402
- `_assignFlipTime()` (*psychopy.visual.rift.Rift* method), 306
- `_blitEyeBuffer()` (*psychopy.visual.nnivs.VisualSystemHD* method), 402
- `_calcEquilateralVertices()` (*psychopy.visual.circle.Circle* static method), 178
- `_calcEquilateralVertices()` (*psychopy.visual.line.Line* static method), 223
- `_calcEquilateralVertices()` (*psychopy.visual.pie.Pie* static method), 256
- `_calcEquilateralVertices()` (*psychopy.visual.polygon.Polygon* static method), 273
- `_calcEquilateralVertices()` (*psychopy.visual.rect.Rect* static method), 296
- `_calcEquilateralVertices()` (*psychopy.visual.shape.ShapeStim* static method), 344
- `_calcPosRendered()` (*psychopy.visual.BufferImageStim* method), 167
- `_calcPosRendered()` (*psychopy.visual.Form* method), 191
- `_calcPosRendered()` (*psychopy.visual.GratingStim* method), 201
- `_calcPosRendered()` (*psychopy.visual.ImageStim* method), 212
- `_calcPosRendered()` (*psychopy.visual.MovieStim* method), 232
- `_calcPosRendered()` (*psychopy.visual.RadialStim* method), 281
- `_calcPosRendered()` (*psychopy.visual.TextBox2* method), 373
- `_calcPosRendered()` (*psychopy.visual.TextStim* method), 381
- `_calcPosRendered()` (*psychopy.visual.VlcMovieStim* method), 391
- `_calcPosRendered()` (*psychopy.visual.circle.Circle* method), 178
- `_calcPosRendered()` (*psychopy.visual.line.Line* method), 223
- `_calcPosRendered()` (*psychopy.visual.pie.Pie* method), 256
- `_calcPosRendered()` (*psychopy.visual.polygon.Polygon* method), 273
- `_calcPosRendered()` (*psychopy.visual.rect.Rect* method), 296
- `_calcPosRendered()` (*psychopy.visual.shape.ShapeStim* method), 344

<i>chopy.visual.shape.ShapeStim method</i>), 344	<i>_cleanEditables ()</i> (psy-
<i>_calcSizeRendered ()</i> (psy-	<i>chopy.visual.nnivs.VisualSystemHD</i>
<i>chopy.visual.BufferImageStim method</i>), 167	402
<i>_calcSizeRendered ()</i> (psychopy.visual. <i>Form</i>	<i>_cleanEditables ()</i> (psychopy.visual. <i>rift.Rift</i>
<i>method</i>), 191	<i>method</i>), 306
<i>_calcSizeRendered ()</i> (psy-	<i>_clip_range ()</i> (psy-
<i>chopy.visual.GratingStim method</i>), 201	<i>chopy.hardware.joystick.XboxController</i>
<i>_calcSizeRendered ()</i> (psychopy.visual. <i>ImageStim</i>	<i>method</i>), 512
<i>method</i>), 212	<i>_closeMedia ()</i> (psychopy.visual. <i>VlcMovieStim</i>
<i>_calcSizeRendered ()</i> (psychopy.visual. <i>MovieStim</i>	<i>method</i>), 391
<i>method</i>), 232	<i>_computeCorners ()</i> (psychopy.visual. <i>BoundingBox</i>
<i>_calcSizeRendered ()</i> (psychopy.visual. <i>RadialStim</i>	<i>method</i>), 158
<i>method</i>), 281	<i>_createDeviceList ()</i> (psy-
<i>_calcSizeRendered ()</i> (psychopy.visual. <i>TextBox2</i>	<i>chopy.iohub.client.ioHubConnection</i>
<i>method</i>), 373	529
<i>_calcSizeRendered ()</i> (psychopy.visual. <i>TextStim</i>	<i>_createItemCtrls ()</i> (psychopy.visual. <i>Form</i>
<i>method</i>), 381	<i>method</i>), 191
<i>_calcSizeRendered ()</i> (psy-	<i>_createOutputArray ()</i> (psy-
<i>chopy.visual.VlcMovieStim method</i>), 391	<i>chopy.data.TrialHandler</i>
<i>_calcSizeRendered ()</i> (psy-	<i>method</i>), 700
<i>chopy.visual.circle.Circle method</i>), 178	<i>_createOutputArray ()</i> (psy-
<i>_calcSizeRendered ()</i> (psychopy.visual. <i>line.Line</i>	<i>chopy.data.TrialHandlerExt</i>
<i>method</i>), 223	<i>method</i>), 709
<i>_calcSizeRendered ()</i> (psychopy.visual. <i>pie.Pie</i>	<i>_createOutputArrayData ()</i> (psy-
<i>method</i>), 256	<i>chopy.data.TrialHandler</i>
<i>_calcSizeRendered ()</i> (psy-	<i>method</i>), 700
<i>chopy.visual.polygon.Polygon method</i>), 273	<i>_createOutputArrayData ()</i> (psy-
<i>_calcSizeRendered ()</i> (psychopy.visual. <i>rect.Rect</i>	<i>chopy.data.TrialHandlerExt</i>
<i>method</i>), 296	<i>method</i>), 709
<i>_calcSizeRendered ()</i> (psy-	<i>_createSequence ()</i> (psychopy. <i>data.TrialHandler</i>
<i>chopy.visual.shape.ShapeStim method</i>), 344	<i>method</i>), 700
<i>_calcVertices ()</i> (psychopy.visual. <i>pie.Pie</i>	<i>_createSequence ()</i> (psy-
<i>method</i>), 256	<i>chopy.data.TrialHandlerExt</i>
<i>_channelCheck ()</i> (psy-	<i>method</i>), 709
<i>chopy.sound.backend_ptb.SoundPTB</i>	<i>_createTexture ()</i> (psy-
<i>method</i>), 443	<i>chopy.visual.BufferImageStim</i>
<i>_channelCheck ()</i> (psy-	<i>method</i>), 167
<i>chopy.sound.backend_sounddevice.SoundDeviceSound</i>	<i>_createTexture ()</i> (psychopy.visual. <i>GratingStim</i>
<i>method</i>), 445	<i>method</i>), 201
<i>_checkCodecSupported ()</i> (psy-	<i>_createTexture ()</i> (psychopy.visual. <i>ImageStim</i>
<i>chopy.sound.AudioClip</i> static	<i>method</i>), 212
<i>method</i>), 454	<i>_createTexture ()</i> (psychopy.visual. <i>RadialStim</i>
<i>_checkFinished ()</i> (psychopy. <i>data.PsiHandler</i>	<i>method</i>), 281
<i>method</i>), 718	<i>_createVAO ()</i> (psychopy.visual. <i>BoxStim</i>
<i>_checkFinished ()</i> (psychopy. <i>data.QuestHandler</i>	<i>method</i>), 160
<i>method</i>), 722	<i>_createVAO ()</i> (psychopy.visual. <i>ObjMeshStim</i>
<i>_checkMatchingSizes ()</i> (psychopy.visual. <i>Window</i>	<i>method</i>), 245
<i>method</i>), 421	<i>_createVAO ()</i> (psychopy.visual. <i>PlaneStim</i>
<i>_checkMatchingSizes ()</i> (psy-	<i>method</i>), 265
<i>chopy.visual.nnivs.VisualSystemHD</i>	<i>_createVAO ()</i> (psychopy.visual. <i>SphereStim</i>
<i>method</i>), 402	<i>method</i>), 357
<i>_checkMatchingSizes ()</i> (psychopy.visual. <i>rift.Rift</i>	<i>_createVLCInstance ()</i> (psy-
<i>method</i>), 306	<i>chopy.visual.VlcMovieStim</i>
<i>_cleanEditables ()</i> (psychopy.visual. <i>Window</i>	<i>method</i>), 391
<i>method</i>), 421	<i>_decodePress ()</i> (psy-
	<i>chopy.hardware.forp.ButtonBox</i>
	<i>class method</i>), 510
	<i>_delete ()</i> (psychopy. <i>hardware.emulator.ResponseEmulator</i>
	<i>method</i>), 505
	<i>_delete ()</i> (psychopy. <i>hardware.emulator.SyncGenerator</i>
	<i>method</i>), 507

<code>_doFit()</code> (<i>psychopy.data.FitCumNormal method</i>), 736	
<code>_doFit()</code> (<i>psychopy.data.FitLogistic method</i>), 735	
<code>_doFit()</code> (<i>psychopy.data.FitNakaRushton method</i>), 735	
<code>_doFit()</code> (<i>psychopy.data.FitWeibull method</i>), 734	
<code>_do_chunk()</code> (<i>psychopy.voicekey.OnsetVoiceKey method</i>), 774	
<code>_drawCtrls()</code> (<i>psychopy.visual.Form method</i>), 191	
<code>_drawDecorations()</code> (<i>psychopy.visual.Form method</i>), 191	
<code>_drawExternalDecorations()</code> (<i>psychopy.visual.Form method</i>), 191	
<code>_drawLUTtoScreen()</code> (<i>psychopy.hardware.crs.bits.BitsPlusPlus method</i>), 472	
<code>_drawLUTtoScreen()</code> (<i>psychopy.hardware.crs.bits.BitsSharp method</i>), 483	
<code>_drawRectangle()</code> (<i>psychopy.visual.MovieStim method</i>), 232	
<code>_drawRectangle()</code> (<i>psychopy.visual.VlcMovieStim method</i>), 391	
<code>_drawTrigtoScreen()</code> (<i>psychopy.hardware.crs.bits.BitsPlusPlus method</i>), 472	
<code>_drawTrigtoScreen()</code> (<i>psychopy.hardware.crs.bits.BitsSharp method</i>), 483	
<code>_endOfFlip()</code> (<i>psychopy.visual.Window method</i>), 421	
<code>_endOfFlip()</code> (<i>psychopy.visual.nnlvs.VisualSystemHD method</i>), 402	
<code>_endOfFlip()</code> (<i>psychopy.visual.rift.Rift method</i>), 306	
<code>_extractStatusEvents()</code> (<i>psychopy.hardware.crs.bits.BitsSharp method</i>), 483	
<code>_freeBuffers()</code> (<i>psychopy.visual.MovieStim method</i>), 232	
<code>_freeBuffers()</code> (<i>psychopy.visual.VlcMovieStim method</i>), 391	
<code>_generateEvents()</code> (<i>psychopy.hardware.forp.ButtonBox method</i>), 510	
<code>_getAllParamNames()</code> (<i>psychopy.data.ExperimentHandler method</i>), 697	
<code>_getDefaultSampleRate()</code> (<i>psychopy.sound.backend_ptb.SoundPTB method</i>), 443	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.BoxStim method</i>), 160	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.BufferImageStim method</i>), 168	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.Form method</i>), 192	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.GratingStim method</i>), 202	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.ImageStim method</i>), 212	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.MovieStim method</i>), 232	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.ObjMeshStim method</i>), 245	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.PlaneStim method</i>), 265	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.RadialStim method</i>), 281	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.SphereStim method</i>), 357	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.TextBox2 method</i>), 373	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.TextStim method</i>), 381	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.circle.Circle method</i>), 178	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.line.Line method</i>), 223	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.pie.Pie method</i>), 257	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.polygon.Polygon method</i>), 273	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.rect.Rect method</i>), 296	
<code>_getDesiredRGB()</code> (<i>psychopy.visual.shape.ShapeStim method</i>), 344	
<code>_getExtraInfo()</code> (<i>psychopy.data.ExperimentHandler method</i>), 697	
<code>_getFrame()</code> (<i>psychopy.visual.Window method</i>), 421	
<code>_getFrame()</code> (<i>psychopy.visual.nnlvs.VisualSystemHD method</i>), 402	
<code>_getFrame()</code> (<i>psychopy.visual.rift.Rift method</i>), 306	
<code>_getHgVersion()</code> (<i>in module psychopy.info</i>), 747	
<code>_getHitboxParams()</code> (<i>psychopy.visual.Slider method</i>), 353	
<code>_getItemHeight()</code> (<i>psychopy.visual.Form method</i>), 192	
<code>_getItemRenderedWidth()</code> (<i>psychopy.visual.Form method</i>), 192	
<code>_getLineParams()</code> (<i>psychopy.visual.Slider method</i>), 353	
<code>_getLoopInfo()</code> (<i>psychopy.data.ExperimentHandler method</i>), 697	
<code>_getMarkerParams()</code> (<i>psychopy.visual.Slider method</i>), 353	
<code>_getPolyAsRendered()</code> (<i>psychopy.visual.BufferImageStim method</i>), 168	
<code>_getPolyAsRendered()</code> (<i>psychopy.visual.Form method</i>), 192	

<i>method</i>), 192		<i>method</i>), 723	
<code>__getPolyAsRendered()</code> (<i>psy-</i>	<code>__intensityDec()</code> (<i>psychopy.data.PsiHandler</i>		
<code>chopy.visual.GratingStim method</code>), 202	<i>method</i>), 718		
<code>__getPolyAsRendered()</code> (<i>psy-</i>	<code>__intensityDec()</code> (<i>psychopy.data.QuestHandler</i>		
<code>chopy.visual.ImageStim method</code>), 212	<i>method</i>), 723		
<code>__getPolyAsRendered()</code> (<i>psy-</i>	<code>__intensityDec()</code> (<i>psychopy.data.QuestPlusHandler</i>		
<code>chopy.visual.MovieStim method</code>), 232	<i>method</i>), 728		
<code>__getPolyAsRendered()</code> (<i>psy-</i>	<code>__intensityDec()</code> (<i>psychopy.data.StairHandler</i>		
<code>chopy.visual.RadialStim method</code>), 281	<i>method</i>), 714		
<code>__getPolyAsRendered()</code> (<i>psychopy.visual.TextBox2</i>	<code>__intensityInc()</code> (<i>psychopy.data.PsiHandler</i>		
<i>method</i>), 373	<i>method</i>), 718		
<code>__getPolyAsRendered()</code> (<i>psychopy.visual.TextStim</i>	<code>__intensityInc()</code> (<i>psychopy.data.QuestHandler</i>		
<i>method</i>), 381	<i>method</i>), 723		
<code>__getPolyAsRendered()</code> (<i>psy-</i>	<code>__intensityInc()</code> (<i>psychopy.data.QuestPlusHandler</i>		
<code>chopy.visual.VlcMovieStim method</code>), 391	<i>method</i>), 728		
<code>__getPolyAsRendered()</code> (<i>psy-</i>	<code>__intensityInc()</code> (<i>psychopy.data.StairHandler</i>		
<code>chopy.visual.circle.Circle method</code>), 178	<i>method</i>), 714		
<code>__getPolyAsRendered()</code> (<i>psychopy.visual.line.Line</i>	<code>__isErrorReply()</code> (<i>psy-</i>		
<i>method</i>), 224	<i>chopy.iohub.client.ioHubConnection</i>		
<code>__getPolyAsRendered()</code> (<i>psychopy.visual.pie.Pie</i>	<i>method</i>), 529		
<i>method</i>), 257	<code>__layout()</code> (<i>psychopy.visual.TextBox2 method</i>), 373		
<code>__getPolyAsRendered()</code> (<i>psy-</i>	<code>__layoutY()</code> (<i>psychopy.visual.Form method</i>), 192		
<code>chopy.visual.polygon.Polygon method</code>), 273	<code>__loadAll()</code> (<i>psychopy.monitors.Monitor method</i>), 762		
<code>__getPolyAsRendered()</code> (<i>psychopy.visual.rect.Rect</i>	<code>__loadMtlLib()</code> (<i>psychopy.visual.ObjMeshStim</i>		
<i>method</i>), 296	<i>method</i>), 245		
<code>__getPolyAsRendered()</code> (<i>psy-</i>	<code>__makeIndices()</code> (<i>psychopy.data.TrialHandler</i>		
<code>chopy.visual.shape.ShapeStim method</code>), 344	<i>method</i>), 700		
<code>__getRegionOfFrame()</code> (<i>psychopy.visual.Window</i>	<code>__makeIndices()</code> (<i>psychopy.data.TrialHandlerExt</i>		
<i>method</i>), 421	<i>method</i>), 710		
<code>__getRegionOfFrame()</code> (<i>psy-</i>	<code>__makeSlider()</code> (<i>psychopy.visual.Form method</i>), 192		
<code>chopy.visual.nnlvs.VisualSystemHD</code>	<code>__makeTextBox()</code> (<i>psychopy.visual.Form method</i>),		
<i>method</i>), 402	192		
<code>__getRegionOfFrame()</code> (<i>psychopy.visual.rift.Rift</i>	<code>__movieFrameToTexture()</code> (<i>psy-</i>		
<i>method</i>), 307	<i>chopy.visual.BufferImageStim method</i>), 168		
<code>__getScrollOffset()</code> (<i>psychopy.visual.Form</i>	<code>__movieFrameToTexture()</code> (<i>psy-</i>		
<i>method</i>), 192	<i>chopy.visual.ImageStim method</i>), 212		
<code>__getShalhexDigest()</code> (<i>in module psychopy.info</i>),	<code>__onCursorKeys()</code> (<i>psychopy.visual.TextBox2</i>		
748	<i>method</i>), 373		
<code>__getStatusLog()</code> (<i>psy-</i>	<code>__onEos()</code> (<i>psychopy.visual.VlcMovieStim method</i>), 391		
<code>chopy.hardware.crs.bits.BitsSharp</code>	<code>__onText()</code> (<i>psychopy.visual.TextBox2 method</i>), 373		
<i>method</i>), 483	<code>__openMedia()</code> (<i>psychopy.visual.VlcMovieStim</i>		
<code>__getSvnVersion()</code> (<i>in module psychopy.info</i>), 748	<i>method</i>), 391		
<code>__getTickParams()</code> (<i>psychopy.visual.Slider method</i>),	<code>__pixelTransfer()</code> (<i>psychopy.visual.MovieStim</i>		
353	<i>method</i>), 232		
<code>__getUserNameUID()</code> (<i>in module psychopy.info</i>), 748	<code>__pixelTransfer()</code> (<i>psychopy.visual.VlcMovieStim</i>		
<code>__getWarpExtents()</code> (<i>psy-</i>	<i>method</i>), 391		
<code>chopy.visual.nnlvs.VisualSystemHD</code>	<code>__prepareMonoFrame()</code> (<i>psychopy.visual.rift.Rift</i>		
<i>method</i>), 402	<i>method</i>), 307		
<code>__granularRating()</code> (<i>psychopy.visual.Slider</i>	<code>__process()</code> (<i>psychopy.voicekey.OnsetVoiceKey</i>		
<i>method</i>), 353	<i>method</i>), 774		
<code>__inRange()</code> (<i>psychopy.visual.Form method</i>), 192	<code>__protectTrigger()</code> (<i>psy-</i>		
<code>__inWaiting()</code> (<i>psychopy.hardware.crs.bits.BitsSharp</i>	<i>chopy.hardware.crs.bits.BitsPlusPlus</i>		
<i>method</i>), 483	<i>method</i>), 472		
<code>__intensity()</code> (<i>psychopy.data.QuestHandler</i>	<code>__protectTrigger()</code> (<i>psy-</i>		
	<i>method</i>), 723		

chopy.hardware.crs.bits.BitsSharp method), 483
 _releaseVLCInstance() (*psy-chopy.visual.VlcMovieStim* method), 391
 _renderFBO() (*psychopy.visual.Window* method), 421
 _renderFBO() (*psy-chopy.visual.nnlvs.VisualSystemHD* method), 402
 _renderFBO() (*psychopy.visual.rift.Rift* method), 307
 _reset() (*psychopy.visual.Aperture* method), 155
 _resolveMSAA() (*psychopy.visual.rift.Rift* method), 307
 _restoreTrigger() (*psy-chopy.hardware.crs.bits.BitsPlusPlus* method), 472
 _restoreTrigger() (*psy-chopy.hardware.crs.bits.BitsSharp* method), 483
 _selectWindow() (*psychopy.visual.BoxStim* method), 160
 _selectWindow() (*psychopy.visual.BufferImageStim* method), 168
 _selectWindow() (*psychopy.visual.Form* method), 193
 _selectWindow() (*psychopy.visual.GratingStim* method), 202
 _selectWindow() (*psychopy.visual.ImageStim* method), 212
 _selectWindow() (*psychopy.visual.MovieStim* method), 232
 _selectWindow() (*psychopy.visual.ObjMeshStim* method), 245
 _selectWindow() (*psychopy.visual.PlaneStim* method), 265
 _selectWindow() (*psychopy.visual.RadialStim* method), 282
 _selectWindow() (*psychopy.visual.SphereStim* method), 357
 _selectWindow() (*psychopy.visual.TextBox2* method), 373
 _selectWindow() (*psychopy.visual.TextStim* method), 382
 _selectWindow() (*psychopy.visual.VlcMovieStim* method), 391
 _selectWindow() (*psychopy.visual.circle.Circle* method), 178
 _selectWindow() (*psychopy.visual.line.Line* method), 224
 _selectWindow() (*psychopy.visual.pie.Pie* method), 257
 _selectWindow() (*psychopy.visual.polygon.Polygon* method), 273
 _selectWindow() (*psychopy.visual.rect.Rect* method), 296
 _selectWindow() (*psychopy.visual.shape.ShapeStim* method), 344
 _sendExperimentInfo() (*psy-chopy.iohub.client.ioHubConnection* method), 529
 _sendSessionInfo() (*psy-chopy.iohub.client.ioHubConnection* method), 529
 _sendToHubServer() (*psy-chopy.iohub.client.ioHubConnection* method), 529
 _set() (*psychopy.visual.BufferImageStim* method), 168
 _set() (*psychopy.visual.Form* method), 193
 _set() (*psychopy.visual.GratingStim* method), 202
 _set() (*psychopy.visual.ImageStim* method), 213
 _set() (*psychopy.visual.MovieStim* method), 232
 _set() (*psychopy.visual.RadialStim* method), 282
 _set() (*psychopy.visual.TextBox2* method), 373
 _set() (*psychopy.visual.TextStim* method), 382
 _set() (*psychopy.visual.VlcMovieStim* method), 391
 _set() (*psychopy.visual.circle.Circle* method), 178
 _set() (*psychopy.visual.line.Line* method), 224
 _set() (*psychopy.visual.pie.Pie* method), 257
 _set() (*psychopy.visual.polygon.Polygon* method), 273
 _set() (*psychopy.visual.rect.Rect* method), 296
 _set() (*psychopy.visual.shape.ShapeStim* method), 344
 _setAperture() (*psychopy.visual.Form* method), 193
 _setBorder() (*psychopy.visual.Form* method), 193
 _setCurrent() (*psychopy.visual.Window* method), 421
 _setCurrent() (*psy-chopy.visual.nnlvs.VisualSystemHD* method), 402
 _setCurrent() (*psychopy.visual.rift.Rift* method), 307
 _setCurrentProcessInfo() (*psy-chopy.info.RunTimeInfo* method), 747
 _setDecorations() (*psychopy.visual.Form* method), 193
 _setExperimentInfo() (*psy-chopy.info.RunTimeInfo* method), 747
 _setHeaders() (*psy-chopy.hardware.crs.bits.BitsPlusPlus* method), 473
 _setHeaders() (*psy-chopy.hardware.crs.bits.BitsSharp* method), 483
 _setPythonInfo() (*psychopy.info.RunTimeInfo* method), 747
 _setQuestion() (*psychopy.visual.Form* method), 193
 _setRadialAttribute() (*psy-chopy.visual.RadialStim* method), 282

<code>_setResponse()</code> (<i>psychopy.visual.Form</i> method), 193	<code>_setupTextureBuffers()</code> (<i>psychopy.visual.MovieStim</i> method), 232
<code>_setScrollBar()</code> (<i>psychopy.visual.Form</i> method), 193	<code>_setupTextureBuffers()</code> (<i>psychopy.visual.VlcMovieStim</i> method), 392
<code>_setSystemInfo()</code> (<i>psychopy.info.RunTimeInfo</i> method), 747	<code>_startHmdFrame()</code> (<i>psychopy.visual.rift.Rift</i> method), 307
<code>_setTextShaders()</code> (<i>psychopy.visual.TextStim</i> method), 382	<code>_startNewPass()</code> (<i>psychopy.data.MultiStairHandler</i> method), 732
<code>_setWindowInfo()</code> (<i>psychopy.info.RunTimeInfo</i> method), 747	<code>_startOfFlip()</code> (<i>psychopy.visual.Window</i> method), 422
<code>_set_baseline()</code> (<i>psychopy.voicekey.OnsetVoiceKey</i> method), 774	<code>_startOfFlip()</code> (<i>psychopy.visual.nnlvs.VisualSystemHD</i> method), 403
<code>_set_defaults()</code> (<i>psychopy.voicekey.OnsetVoiceKey</i> method), 774	<code>_startOfFlip()</code> (<i>psychopy.visual.rift.Rift</i> method), 308
<code>_set_signaler()</code> (<i>psychopy.voicekey.OnsetVoiceKey</i> method), 774	<code>_startServer()</code> (<i>psychopy.iohub.client.ioHubConnection</i> method), 529
<code>_set_source()</code> (<i>psychopy.voicekey.OnsetVoiceKey</i> method), 774	<code>_statusBox()</code> (<i>psychopy.hardware.crs.bits.BitsSharp</i> method), 483
<code>_set_tables()</code> (<i>psychopy.voicekey.OnsetVoiceKey</i> method), 774	<code>_statusDisable()</code> (<i>psychopy.hardware.crs.bits.BitsSharp</i> method), 484
<code>_set_tstate_lock()</code> (<i>psychopy.hardware.emulator.ResponseEmulator</i> method), 505	<code>_statusEnable()</code> (<i>psychopy.hardware.crs.bits.BitsSharp</i> method), 484
<code>_set_tstate_lock()</code> (<i>psychopy.hardware.emulator.SyncGenerator</i> method), 507	<code>_statusLog()</code> (<i>psychopy.hardware.crs.bits.BitsSharp</i> method), 484
<code>_setupEyeBuffers()</code> (<i>psychopy.visual.nnlvs.VisualSystemHD</i> method), 403	<code>_syncDeviceState()</code> (<i>psychopy.iohub.client.keyboard.Keyboard</i> method), 531
<code>_setupFrameBuffer()</code> (<i>psychopy.visual.rift.Rift</i> method), 307	<code>_terminate()</code> (<i>psychopy.data.MultiStairHandler</i> method), 732
<code>_setupGL()</code> (<i>psychopy.visual.Window</i> method), 421	<code>_terminate()</code> (<i>psychopy.data.PsiHandler</i> method), 718
<code>_setupGL()</code> (<i>psychopy.visual.nnlvs.VisualSystemHD</i> method), 403	<code>_terminate()</code> (<i>psychopy.data.QuestHandler</i> method), 723
<code>_setupGL()</code> (<i>psychopy.visual.rift.Rift</i> method), 307	<code>_terminate()</code> (<i>psychopy.data.QuestPlusHandler</i> method), 728
<code>_setupGamma()</code> (<i>psychopy.visual.Window</i> method), 421	<code>_terminate()</code> (<i>psychopy.data.StairHandler</i> method), 714
<code>_setupGamma()</code> (<i>psychopy.visual.nnlvs.VisualSystemHD</i> method), 403	<code>_terminate()</code> (<i>psychopy.data.TrialHandler</i> method), 700
<code>_setupGamma()</code> (<i>psychopy.visual.rift.Rift</i> method), 307	<code>_terminate()</code> (<i>psychopy.data.TrialHandler2</i> method), 705
<code>_setupLensCorrection()</code> (<i>psychopy.visual.nnlvs.VisualSystemHD</i> method), 403	<code>_terminate()</code> (<i>psychopy.data.TrialHandlerExt</i> method), 710
<code>_setupShaders()</code> (<i>psychopy.hardware.crs.bits.BitsPlusPlus</i> method), 473	<code>_tessellate()</code> (<i>psychopy.visual.line.Line</i> method), 224
<code>_setupShaders()</code> (<i>psychopy.hardware.crs.bits.BitsSharp</i> method), 483	<code>_tessellate()</code> (<i>psychopy.visual.shape.ShapeStim</i> method), 344
	<code>_updateEverything()</code> (<i>psychopy.visual.RadialStim</i> method), 282
	<code>_updateList()</code> (<i>psychopy.visual.BoxStim</i> method),

- `_updateList ()` (*psychopy.visual.BufferImageStim method*), 168
- `_updateList ()` (*psychopy.visual.Form method*), 193
- `_updateList ()` (*psychopy.visual.GratingStim method*), 202
- `_updateList ()` (*psychopy.visual.ImageStim method*), 213
- `_updateList ()` (*psychopy.visual.MovieStim method*), 232
- `_updateList ()` (*psychopy.visual.ObjMeshStim method*), 245
- `_updateList ()` (*psychopy.visual.PlaneStim method*), 265
- `_updateList ()` (*psychopy.visual.RadialStim method*), 282
- `_updateList ()` (*psychopy.visual.SphereStim method*), 357
- `_updateList ()` (*psychopy.visual.TextBox2 method*), 373
- `_updateList ()` (*psychopy.visual.TextStim method*), 382
- `_updateList ()` (*psychopy.visual.VlcMovieStim method*), 392
- `_updateList ()` (*psychopy.visual.circle.Circle method*), 178
- `_updateList ()` (*psychopy.visual.line.Line method*), 224
- `_updateList ()` (*psychopy.visual.pie.Pie method*), 257
- `_updateList ()` (*psychopy.visual.polygon.Polygon method*), 273
- `_updateList ()` (*psychopy.visual.rect.Rect method*), 296
- `_updateList ()` (*psychopy.visual.shape.ShapeStim method*), 344
- `_updateListShaders ()` (*psychopy.visual.BufferImageStim method*), 168
- `_updateListShaders ()` (*psychopy.visual.GratingStim method*), 202
- `_updateListShaders ()` (*psychopy.visual.ImageStim method*), 213
- `_updateListShaders ()` (*psychopy.visual.RadialStim method*), 282
- `_updateListShaders ()` (*psychopy.visual.TextStim method*), 382
- `_updateMaskCoords ()` (*psychopy.visual.RadialStim method*), 282
- `_updatePerfStats ()` (*psychopy.visual.rift.Rift method*), 308
- `_updateProjectionMatrix ()` (*psychopy.visual.rift.Rift method*), 308
- `_updateTextureCoords ()` (*psychopy.visual.RadialStim method*), 282
- `_updateVertices ()` (*psychopy.visual.Aperture method*), 156
- `_updateVertices ()` (*psychopy.visual.BufferImageStim method*), 168
- `_updateVertices ()` (*psychopy.visual.Form method*), 193
- `_updateVertices ()` (*psychopy.visual.GratingStim method*), 202
- `_updateVertices ()` (*psychopy.visual.ImageStim method*), 213
- `_updateVertices ()` (*psychopy.visual.MovieStim method*), 232
- `_updateVertices ()` (*psychopy.visual.RadialStim method*), 282
- `_updateVertices ()` (*psychopy.visual.TextBox2 method*), 374
- `_updateVertices ()` (*psychopy.visual.TextStim method*), 382
- `_updateVertices ()` (*psychopy.visual.VlcMovieStim method*), 392
- `_updateVertices ()` (*psychopy.visual.circle.Circle method*), 178
- `_updateVertices ()` (*psychopy.visual.line.Line method*), 224
- `_updateVertices ()` (*psychopy.visual.pie.Pie method*), 257
- `_updateVertices ()` (*psychopy.visual.polygon.Polygon method*), 273
- `_updateVertices ()` (*psychopy.visual.rect.Rect method*), 297
- `_updateVertices ()` (*psychopy.visual.shape.ShapeStim method*), 345
- `_updateVerticesBase ()` (*psychopy.visual.RadialStim method*), 282
- `_waitToBeginHmdFrame ()` (*psychopy.visual.rift.Rift method*), 308

A

- `abort ()` (*psychopy.data.ExperimentHandler method*), 697
- `accumQuat ()` (*in module psychopy.tools.mathtools*), 655
- `active ()` (*psychopy.sound.AudioDeviceStatus property*), 462
- Adaptive staircase, 30
- `add ()` (*psychopy.clock.Clock method*), 152
- `add ()` (*psychopy.core.Clock method*), 149
- `addCharAtCaret ()` (*psychopy.visual.TextBox2 method*), 374
- `addData ()` (*psychopy.data.ExperimentHandler method*), 697
- `addData ()` (*psychopy.data.MultiStairHandler method*), 732
- `addData ()` (*psychopy.data.PsiHandler method*), 718

- addData () (*psychopy.data.QuestHandler* method), 723
- addData () (*psychopy.data.QuestPlusHandler* method), 728
- addData () (*psychopy.data.StairHandler* method), 714
- addData () (*psychopy.data.TrialHandler* method), 700
- addData () (*psychopy.data.TrialHandler2* method), 705
- addData () (*psychopy.data.TrialHandlerExt* method), 710
- addDataToExp () (*psychopy.visual.Form* method), 194
- addEditable () (*psychopy.visual.nnlvs.VisualSystemHD* method), 403
- addEditable () (*psychopy.visual.rift.Rift* method), 308
- addEditable () (*psychopy.visual.Window* method), 422
- addField () (*psychopy.gui.Dlg* method), 745
- addFixedField () (*psychopy.gui.Dlg* method), 745
- addLevel () (in module *psychopy.logging*), 755
- addLoop () (*psychopy.data.ExperimentHandler* method), 697
- addOtherData () (*psychopy.data.MultiStairHandler* method), 732
- addOtherData () (*psychopy.data.PsiHandler* method), 718
- addOtherData () (*psychopy.data.QuestHandler* method), 723
- addOtherData () (*psychopy.data.QuestPlusHandler* method), 728
- addOtherData () (*psychopy.data.StairHandler* method), 714
- addResponse () (*psychopy.data.MultiStairHandler* method), 732
- addResponse () (*psychopy.data.PsiHandler* method), 718
- addResponse () (*psychopy.data.QuestHandler* method), 723
- addResponse () (*psychopy.data.QuestPlusHandler* method), 728
- addResponse () (*psychopy.data.StairHandler* method), 714
- addTarget () (*psychopy.logging._Logger* method), 754
- addTime () (*psychopy.clock.Clock* method), 152
- addTime () (*psychopy.core.Clock* method), 149
- addTrialHandlerRecord () (*psychopy.iohub.client.ioHubConnection* method), 527
- AdvAudioCapture (class in *psychopy.microphone*), 757
- alignHoriz (*psychopy.visual.TextStim* attribute), 382
- alignment () (*psychopy.visual.TextBox2* property), 374
- alignText (*psychopy.visual.TextStim* attribute), 382
- alignTo () (in module *psychopy.tools.mathtools*), 653
- alignTo () (*psychopy.visual.RigidBodyPose* method), 336
- alignVert (*psychopy.visual.TextStim* attribute), 382
- alpha () (*psychopy.colors.Color* property), 694
- ambientColor () (*psychopy.visual.BlinnPhongMaterial* property), 253
- ambientColor () (*psychopy.visual.LightSource* property), 220
- ambientLight () (*psychopy.visual.nnlvs.VisualSystemHD* property), 403
- ambientLight () (*psychopy.visual.rift.Rift* property), 308
- ambientLight () (*psychopy.visual.Window* property), 422
- ambientRGB () (*psychopy.visual.BlinnPhongMaterial* property), 253
- ambientRGB () (*psychopy.visual.LightSource* property), 221
- amplifier () (*psychopy.hardware.brainproducts.RemoteControlServer* property), 467
- anchor () (*psychopy.layout.Vertices* property), 753
- anchor () (*psychopy.visual.Aperture* property), 156
- anchor () (*psychopy.visual.BoxStim* property), 160
- anchor () (*psychopy.visual.BufferImageStim* property), 168
- anchor () (*psychopy.visual.Form* property), 194
- anchor () (*psychopy.visual.GratingStim* property), 202
- anchor () (*psychopy.visual.ImageStim* property), 213
- anchor () (*psychopy.visual.MovieStim* property), 232
- anchor () (*psychopy.visual.ObjMeshStim* property), 246
- anchor () (*psychopy.visual.PlaneStim* property), 265
- anchor () (*psychopy.visual.RadialStim* property), 282
- anchor () (*psychopy.visual.SphereStim* property), 357
- anchor () (*psychopy.visual.TextBox2* property), 374
- anchor () (*psychopy.visual.VlcMovieStim* property), 392
- anchorAdjust () (*psychopy.layout.Vertices* property), 753
- anchorHoriz (*psychopy.visual.TextStim* attribute), 382
- anchorVert (*psychopy.visual.TextStim* attribute), 382
- angle_x (*psychopy.iohub.devices.eyetracker.FixationStartEvent* attribute), 557
- angle_x (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 550, 554
- angle_x (*psychopy.iohub.devices.eyetracker.SaccadeStartEvent* attribute), 559
- angle_y (*psychopy.iohub.devices.eyetracker.FixationStartEvent* attribute), 557

- angle_y (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent attribute*), 550, 554
- angle_y (*psychopy.iohub.devices.eyetracker.SaccadeStartEvent attribute*), 559
- angleTo () (*in module psychopy.tools.mathtools*), 641
- angularCycles (*psychopy.visual.RadialStim attribute*), 282
- angularPhase (*psychopy.visual.RadialStim attribute*), 282
- angularRes (*psychopy.visual.RadialStim attribute*), 282
- antialias (*psychopy.visual.TextStim attribute*), 382
- Aperture (*class in psychopy.visual*), 155
- apodize () (*in module psychopy.voicekey*), 776
- append () (*psychopy.sound.AudioClip method*), 454
- applyEyeTransform () (*psychopy.visual.nnlvs.VisualSystemHD method*), 403
- applyEyeTransform () (*psychopy.visual.rift.Rift method*), 308
- applyEyeTransform () (*psychopy.visual.Window method*), 422
- applyMatrix () (*in module psychopy.tools.mathtools*), 668
- applyQuat () (*in module psychopy.tools.mathtools*), 657
- array2image () (*in module psychopy.tools.imagetools*), 634
- articulate () (*in module psychopy.tools.mathtools*), 649
- asMono () (*psychopy.sound.AudioClip method*), 454
- aspect () (*psychopy.visual.nnlvs.VisualSystemHD property*), 404
- aspect () (*psychopy.visual.rift.Rift property*), 309
- aspect () (*psychopy.visual.Window property*), 422
- aspirate () (*psychopy.hardware.qmix.Pump method*), 521
- at () (*psychopy.visual.RigidBodyPose property*), 336
- attach () (*in module psychopy.tools.gltools*), 602
- attachObjectARB () (*in module psychopy.tools.gltools*), 594
- attachShader () (*in module psychopy.tools.gltools*), 593
- attenuation () (*psychopy.visual.LightSource property*), 221
- AudioClip (*class in psychopy.sound*), 453
- AudioDeviceInfo (*class in psychopy.sound*), 459
- AudioDeviceStatus (*class in psychopy.sound*), 461
- audioLatencyMode () (*psychopy.sound.Microphone property*), 449
- audioLib () (*psychopy.sound.AudioDeviceInfo property*), 460
- audioLib () (*psychopy.sound.AudioDeviceStatus property*), 462
- autoDraw (*psychopy.visual.Aperture attribute*), 156
- autoDraw (*psychopy.visual.BufferImageStim attribute*), 168
- autoDraw (*psychopy.visual.circle.Circle attribute*), 178
- autoDraw (*psychopy.visual.Form attribute*), 194
- autoDraw (*psychopy.visual.GratingStim attribute*), 202
- autoDraw (*psychopy.visual.ImageStim attribute*), 213
- autoDraw (*psychopy.visual.line.Line attribute*), 224
- autoDraw (*psychopy.visual.MovieStim attribute*), 232
- autoDraw (*psychopy.visual.pie.Pie attribute*), 257
- autoDraw (*psychopy.visual.polygon.Polygon attribute*), 273
- autoDraw (*psychopy.visual.RadialStim attribute*), 282
- autoDraw (*psychopy.visual.rect.Rect attribute*), 297
- autoDraw (*psychopy.visual.shape.ShapeStim attribute*), 345
- autoDraw (*psychopy.visual.TextBox2 attribute*), 374
- autoDraw (*psychopy.visual.TextStim attribute*), 382
- autoDraw (*psychopy.visual.VlcMovieStim attribute*), 392
- autoLog (*psychopy.visual.Aperture attribute*), 156
- autoLog (*psychopy.visual.BufferImageStim attribute*), 168
- autoLog (*psychopy.visual.circle.Circle attribute*), 178
- autoLog (*psychopy.visual.Form attribute*), 194
- autoLog (*psychopy.visual.GratingStim attribute*), 202
- autoLog (*psychopy.visual.ImageStim attribute*), 213
- autoLog (*psychopy.visual.line.Line attribute*), 224
- autoLog (*psychopy.visual.MovieStim attribute*), 233
- autoLog (*psychopy.visual.pie.Pie attribute*), 257
- autoLog (*psychopy.visual.polygon.Polygon attribute*), 273
- autoLog (*psychopy.visual.RadialStim attribute*), 282
- autoLog (*psychopy.visual.rect.Rect attribute*), 297
- autoLog (*psychopy.visual.shape.ShapeStim attribute*), 345
- autoLog (*psychopy.visual.TextBox2 attribute*), 374
- autoLog (*psychopy.visual.TextStim attribute*), 382
- autoLog (*psychopy.visual.VlcMovieStim attribute*), 392
- autoStart () (*psychopy.visual.MovieStim property*), 233
- autoStart () (*psychopy.visual.VlcMovieStim property*), 392
- average_angle_x (*psychopy.iohub.devices.eyetracker.FixationEndEvent attribute*), 558
- average_angle_x (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent attribute*), 561
- average_angle_y (*psychopy.iohub.devices.eyetracker.FixationEndEvent attribute*), 559
- average_angle_y (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent attribute*), 559

<i>attribute</i>), 561	<i>backColor()</i> (<i>psychopy.visual.RadialStim</i> property), 283
<i>average_gaze_x</i> (<i>psychopy.iohub.devices.eyetracker.FixationEndEvent</i> attribute), 540, 558, 579	<i>backColor()</i> (<i>psychopy.visual.rect.Rect</i> property), 297
<i>average_gaze_x</i> (<i>psychopy.iohub.devices.eyetracker.SaccadeEndEvent</i> attribute), 561	<i>backColor()</i> (<i>psychopy.visual.shape.ShapeStim</i> property), 345
<i>average_gaze_y</i> (<i>psychopy.iohub.devices.eyetracker.FixationEndEvent</i> attribute), 540, 558, 579	<i>backColor()</i> (<i>psychopy.visual.SphereStim</i> property), 357
<i>average_gaze_y</i> (<i>psychopy.iohub.devices.eyetracker.SaccadeEndEvent</i> attribute), 561	<i>backColor()</i> (<i>psychopy.visual.TextBox2</i> property), 374
<i>average_pupil_measure1_type</i> (<i>psychopy.iohub.devices.eyetracker.FixationEndEvent</i> attribute), 559	<i>backColorSpace()</i> (<i>psychopy.visual.BoxStim</i> property), 160
<i>average_pupil_measure1_type</i> (<i>psychopy.iohub.devices.eyetracker.SaccadeEndEvent</i> attribute), 561	<i>backColorSpace()</i> (<i>psychopy.visual.BufferImageStim</i> property), 168
<i>average_pupil_measure_1</i> (<i>psychopy.iohub.devices.eyetracker.FixationEndEvent</i> attribute), 559	<i>backColorSpace()</i> (<i>psychopy.visual.circle.Circle</i> property), 179
<i>average_pupil_measure_1</i> (<i>psychopy.iohub.devices.eyetracker.SaccadeEndEvent</i> attribute), 561	<i>backColorSpace()</i> (<i>psychopy.visual.Form</i> property), 194
<i>average_velocity_xy</i> (<i>psychopy.iohub.devices.eyetracker.FixationEndEvent</i> attribute), 559	<i>backColorSpace()</i> (<i>psychopy.visual.GratingStim</i> property), 203
<i>average_velocity_xy</i> (<i>psychopy.iohub.devices.eyetracker.SaccadeEndEvent</i> attribute), 561	<i>backColorSpace()</i> (<i>psychopy.visual.ImageStim</i> property), 213
<i>average_velocity_xy</i> (<i>psychopy.iohub.devices.eyetracker.FixationEndEvent</i> attribute), 559	<i>backColorSpace()</i> (<i>psychopy.visual.line.Line</i> property), 224
<i>average_velocity_xy</i> (<i>psychopy.iohub.devices.eyetracker.SaccadeEndEvent</i> attribute), 561	<i>backColorSpace()</i> (<i>psychopy.visual.MovieStim</i> property), 233
	<i>backColorSpace()</i> (<i>psychopy.visual.ObjMeshStim</i> property), 246
	<i>backColorSpace()</i> (<i>psychopy.visual.pie.Pie</i> property), 257
B	<i>backColorSpace()</i> (<i>psychopy.visual.PlaneStim</i> property), 265
<i>backColor()</i> (<i>psychopy.visual.BoxStim</i> property), 160	<i>backColorSpace()</i> (<i>psychopy.visual.polygon.Polygon</i> property), 273
<i>backColor()</i> (<i>psychopy.visual.BufferImageStim</i> property), 168	<i>backColorSpace()</i> (<i>psychopy.visual.RadialStim</i> property), 283
<i>backColor()</i> (<i>psychopy.visual.circle.Circle</i> property), 179	<i>backColorSpace()</i> (<i>psychopy.visual.rect.Rect</i> property), 297
<i>backColor()</i> (<i>psychopy.visual.Form</i> property), 194	<i>backColorSpace()</i> (<i>psychopy.visual.shape.ShapeStim</i> property), 345
<i>backColor()</i> (<i>psychopy.visual.GratingStim</i> property), 202	<i>backColorSpace()</i> (<i>psychopy.visual.SphereStim</i> property), 357
<i>backColor()</i> (<i>psychopy.visual.ImageStim</i> property), 213	<i>backColorSpace()</i> (<i>psychopy.visual.TextBox2</i> property), 374
<i>backColor()</i> (<i>psychopy.visual.line.Line</i> property), 224	<i>backRGB()</i> (<i>psychopy.visual.BoxStim</i> property), 160
<i>backColor()</i> (<i>psychopy.visual.MovieStim</i> property), 233	<i>backRGB()</i> (<i>psychopy.visual.BufferImageStim</i> property), 168
<i>backColor()</i> (<i>psychopy.visual.ObjMeshStim</i> property), 246	<i>backRGB()</i> (<i>psychopy.visual.circle.Circle</i> property), 179
<i>backColor()</i> (<i>psychopy.visual.pie.Pie</i> property), 257	<i>backRGB()</i> (<i>psychopy.visual.Form</i> property), 194
<i>backColor()</i> (<i>psychopy.visual.PlaneStim</i> property), 265	<i>backRGB()</i> (<i>psychopy.visual.GratingStim</i> property),
<i>backColor()</i> (<i>psychopy.visual.polygon.Polygon</i> property), 273	

- 203
- backRGB () (*psychopy.visual.ImageStim* property), 213
- backRGB () (*psychopy.visual.line.Line* property), 224
- backRGB () (*psychopy.visual.MovieStim* property), 233
- backRGB () (*psychopy.visual.ObjMeshStim* property), 246
- backRGB () (*psychopy.visual.pie.Pie* property), 257
- backRGB () (*psychopy.visual.PlaneStim* property), 265
- backRGB () (*psychopy.visual.polygon.Polygon* property), 273
- backRGB () (*psychopy.visual.RadialStim* property), 283
- backRGB () (*psychopy.visual.rect.Rect* property), 297
- backRGB () (*psychopy.visual.shape.ShapeStim* property), 345
- backRGB () (*psychopy.visual.SphereStim* property), 357
- backRGB () (*psychopy.visual.TextBox2* property), 374
- bandpass () (in module *psychopy.voicekey*), 775
- bank () (*psychopy.sound.Microphone* method), 449
- beep () (*psychopy.hardware.crs.bits.BitsSharp* method), 484
- begin () (*psychopy.visual.BlinnPhongMaterial* method), 253
- beginQuery () (in module *psychopy.tools.gltools*), 599
- beta () (*psychopy.data.QuestHandler* property), 723
- bindTexture () (in module *psychopy.tools.gltools*), 607
- bindVBO () (in module *psychopy.tools.gltools*), 614
- BinocularEyeSampleEvent (class in *psychopy.iohub.devices.eyetracker*), 539, 550, 555, 570, 578
- bisector () (in module *psychopy.tools.mathtools*), 642
- BitsPlusPlus (class in *psychopy.hardware.crs.bits*), 472
- BitsSharp (class in *psychopy.hardware.crs.bits*), 478
- blendmode (*psychopy.visual.GratingStim* attribute), 203
- blendMode (*psychopy.visual.nnlvs.VisualSystemHD* attribute), 404
- blendmode (*psychopy.visual.RadialStim* attribute), 283
- blendMode (*psychopy.visual.rift.Rift* attribute), 309
- blendMode (*psychopy.visual.Window* attribute), 423
- BlinkEndEvent (class in *psychopy.iohub.devices.eyetracker*), 561
- BlinkStartEvent (class in *psychopy.iohub.devices.eyetracker*), 561
- BlinnPhongMaterial (class in *psychopy.visual*), 251
- blitFBO () (in module *psychopy.tools.gltools*), 602
- bold (*psychopy.visual.TextStim* attribute), 382
- bootStraps () (in module *psychopy.data*), 737
- borderColor () (*psychopy.visual.BoxStim* property), 160
- borderColor () (*psychopy.visual.BufferImageStim* property), 168
- borderColor () (*psychopy.visual.circle.Circle* property), 179
- borderColor () (*psychopy.visual.Form* property), 194
- borderColor () (*psychopy.visual.GratingStim* property), 203
- borderColor () (*psychopy.visual.ImageStim* property), 213
- borderColor () (*psychopy.visual.line.Line* property), 224
- borderColor () (*psychopy.visual.MovieStim* property), 233
- borderColor () (*psychopy.visual.ObjMeshStim* property), 246
- borderColor () (*psychopy.visual.pie.Pie* property), 257
- borderColor () (*psychopy.visual.PlaneStim* property), 265
- borderColor () (*psychopy.visual.polygon.Polygon* property), 274
- borderColor () (*psychopy.visual.RadialStim* property), 283
- borderColor () (*psychopy.visual.rect.Rect* property), 297
- borderColor () (*psychopy.visual.shape.ShapeStim* property), 345
- borderColor () (*psychopy.visual.SphereStim* property), 357
- borderColor () (*psychopy.visual.Slider* property), 353
- borderColor () (*psychopy.visual.SphereStim* property), 357
- borderColor () (*psychopy.visual.TextBox2* property), 374
- borderColorSpace () (*psychopy.visual.BoxStim* property), 160
- borderColorSpace () (*psychopy.visual.BufferImageStim* property), 168
- borderColorSpace () (*psychopy.visual.circle.Circle* property), 179
- borderColorSpace () (*psychopy.visual.Form* property), 194
- borderColorSpace () (*psychopy.visual.GratingStim* property), 203
- borderColorSpace () (*psychopy.visual.ImageStim* property), 213
- borderColorSpace () (*psychopy.visual.line.Line* property), 224
- borderColorSpace () (*psychopy.visual.MovieStim* property), 233
- borderColorSpace () (*psychopy.visual.ObjMeshStim* property), 246
- borderColorSpace () (*psychopy.visual.pie.Pie* property), 257
- borderColorSpace () (*psychopy.visual.PlaneStim* property), 265
- borderColorSpace () (*psychopy.visual.polygon.Polygon* property), 274
- borderColorSpace () (*psychopy.visual.RadialStim* property), 283
- borderColorSpace () (*psychopy.visual.rect.Rect* property), 297
- borderColorSpace () (*psychopy.visual.shape.ShapeStim* property), 345
- borderColorSpace () (*psychopy.visual.SphereStim* property), 357

- borderColorSpace () (*psychopy.visual.TextBox2 property*), 374
- borderRGB () (*psychopy.visual.BoxStim property*), 160
- borderRGB () (*psychopy.visual.BufferImageStim property*), 169
- borderRGB () (*psychopy.visual.circle.Circle property*), 179
- borderRGB () (*psychopy.visual.Form property*), 194
- borderRGB () (*psychopy.visual.GratingStim property*), 203
- borderRGB () (*psychopy.visual.ImageStim property*), 213
- borderRGB () (*psychopy.visual.line.Line property*), 224
- borderRGB () (*psychopy.visual.MovieStim property*), 233
- borderRGB () (*psychopy.visual.ObjMeshStim property*), 246
- borderRGB () (*psychopy.visual.pie.Pie property*), 257
- borderRGB () (*psychopy.visual.PlaneStim property*), 265
- borderRGB () (*psychopy.visual.polygon.Polygon property*), 274
- borderRGB () (*psychopy.visual.RadialStim property*), 283
- borderRGB () (*psychopy.visual.rect.Rect property*), 297
- borderRGB () (*psychopy.visual.shape.ShapeStim property*), 345
- borderRGB () (*psychopy.visual.SphereStim property*), 357
- borderRGB () (*psychopy.visual.TextBox2 property*), 374
- BoundingBox (*class in psychopy.visual*), 158
- boundingBox () (*psychopy.visual.TextStim property*), 382
- bounds () (*psychopy.visual.RigidBodyPose property*), 336
- BoxStim (*class in psychopy.visual*), 159
- BufferImageStim (*class in psychopy.visual*), 166
- bufferSize () (*psychopy.sound.AudioDeviceStatus property*), 462
- butter2d_bp () (*in module psychopy.visual.filters*), 741
- butter2d_hp () (*in module psychopy.visual.filters*), 741
- butter2d_lp () (*in module psychopy.visual.filters*), 741
- butter2d_lp_elliptic () (*in module psychopy.visual.filters*), 741
- ButtonBox (*class in psychopy.hardware.forp*), 510
- C**
- cacheMessageEvent () (*psy-*
- chopy.iohub.client.ioHubConnection method*), 527
- calcEyePoses () (*psychopy.visual.rift.Rift method*), 309
- calculateNextIntensity () (*psychopy.data.PsiHandler method*), 718
- calculateNextIntensity () (*psychopy.data.QuestHandler method*), 723
- calculateNextIntensity () (*psychopy.data.QuestPlusHandler method*), 728
- calculateNextIntensity () (*psychopy.data.StairHandler method*), 715
- calculateVertexNormals () (*in module psychopy.tools.gltools*), 631
- calibrate () (*psychopy.hardware.qmix.Pump method*), 521
- calibrateZero () (*psychopy.hardware.crs.colorcal.ColorCAL method*), 504
- callOnFlip () (*psychopy.visual.nnlvs.VisualSystemHD method*), 404
- callOnFlip () (*psychopy.visual.rift.Rift method*), 310
- callOnFlip () (*psychopy.visual.Window method*), 423
- captureStartTime () (*psychopy.sound.AudioDeviceStatus property*), 462
- cart2pol () (*in module psychopy.tools.coordinatetools*), 587
- cart2sph () (*in module psychopy.tools.coordinatetools*), 587
- categorical () (*psychopy.visual.Slider property*), 353
- changeProjection () (*psychopy.visual.windowwarp.Warper method*), 438
- channels () (*psychopy.sound.AudioClip property*), 454
- char () (*psychopy.iohub.client.keyboard.KeyboardPress property*), 533
- char () (*psychopy.iohub.client.keyboard.KeyboardRelease property*), 534
- checkConfig () (*psychopy.hardware.crs.bits.BitsSharp method*), 484
- checkOK () (*psychopy.hardware.minolta.CS100A method*), 515
- checkOK () (*psychopy.hardware.minolta.LS100 method*), 517
- cielab2rgb () (*in module psychopy.tools.colorspectools*), 581
- cielch2rgb () (*in module psychopy.tools.colorspectools*), 582

- Circle (class in *psychopy.visual.circle*), 176
- clear () (*psychopy.sound.Microphone* method), 450
- clear () (*psychopy.visual.BoundingBox* method), 158
- clear () (*psychopy.visual.TextBox2* method), 374
- clearBuffer () (*psychopy.hardware.forp.ButtonBox* method), 510
- clearBuffer () (*psychopy.visual.nnlvs.VisualSystemHD* method), 404
- clearBuffer () (*psychopy.visual.rift.Rift* method), 310
- clearBuffer () (*psychopy.visual.Window* method), 423
- clearEvents () (in module *psychopy.event*), 739
- clearEvents () (*psychopy.iohub.client.ioHubConnection* method), 526
- clearEvents () (*psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker* method), 536
- clearEvents () (*psychopy.iohub.devices.eyetracker.hw.mouse.EyeTracker* method), 575
- clearEvents () (*psychopy.iohub.devices.eyetracker.hw.tobii.EyeTracker* method), 568
- clearFaultState () (*psychopy.hardware.qmix.Pump* method), 521
- clearMemory () (*psychopy.hardware.minolta.CS100A* method), 515
- clearMemory () (*psychopy.hardware.minolta.LS100* method), 517
- clearShouldRecenterFlag () (*psychopy.visual.rift.Rift* method), 310
- clearStatus () (*psychopy.hardware.forp.ButtonBox* method), 510
- clearTextures () (*psychopy.visual.BufferImageStim* method), 169
- clearTextures () (*psychopy.visual.GratingStim* method), 203
- clearTextures () (*psychopy.visual.ImageStim* method), 213
- clearTextures () (*psychopy.visual.RadialStim* method), 283
- clickReset () (*psychopy.event.Mouse* method), 738
- Clock (class in *psychopy.clock*), 152
- Clock (class in *psychopy.core*), 149
- clock () (*psychopy.hardware.crs.bits.BitsSharp* method), 484
- close () (*psychopy.data.ExperimentHandler* method), 697
- close () (*psychopy.hardware.brainproducts.RemoteControlServer* method), 468
- close () (*psychopy.sound.Microphone* method), 450
- close () (*psychopy.visual.nnlvs.VisualSystemHD* method), 405
- close () (*psychopy.visual.rift.Rift* method), 310
- close () (*psychopy.visual.Window* method), 423
- closeShape (*psychopy.visual.circle.Circle* attribute), 179
- closeShape (*psychopy.visual.line.Line* attribute), 224
- closeShape (*psychopy.visual.pie.Pie* attribute), 257
- closeShape (*psychopy.visual.polygon.Polygon* attribute), 274
- closeShape (*psychopy.visual.rect.Rect* attribute), 297
- closeShape (*psychopy.visual.shape.ShapeStim* attribute), 345
- cm () (*psychopy.layout.Position* property), 750
- cm () (*psychopy.layout.Size* property), 751
- cm () (*psychopy.layout.Vector* property), 749
- cm () (*psychopy.layout.Vertices* property), 753
- cm2deg () (in module *psychopy.tools.monitorunittools*), 678
- cm2pix () (in module *psychopy.tools.monitorunittools*), 678
- coherence (*psychopy.visual.DotStim* attribute), 187
- Color (class in *psychopy.colors*), 694
- color (*psychopy.visual.circle.Circle* attribute), 179
- color (*psychopy.visual.pie.Pie* attribute), 257
- color (*psychopy.visual.polygon.Polygon* attribute), 274
- color (*psychopy.visual.rect.Rect* attribute), 297
- color (*psychopy.visual.shape.ShapeStim* attribute), 345
- color () (*psychopy.visual.BoxStim* property), 160
- color () (*psychopy.visual.BufferImageStim* property), 169
- color () (*psychopy.visual.Form* property), 194
- color () (*psychopy.visual.GratingStim* property), 203
- color () (*psychopy.visual.ImageStim* property), 213
- color () (*psychopy.visual.line.Line* property), 224
- color () (*psychopy.visual.MovieStim* property), 233
- color () (*psychopy.visual.nnlvs.VisualSystemHD* property), 405
- color () (*psychopy.visual.ObjMeshStim* property), 246
- color () (*psychopy.visual.PlaneStim* property), 265
- color () (*psychopy.visual.RadialStim* property), 283
- color () (*psychopy.visual.rift.Rift* property), 310
- color () (*psychopy.visual.SphereStim* property), 358
- color () (*psychopy.visual.TextBox2* property), 374
- color () (*psychopy.visual.TextStim* property), 382
- color () (*psychopy.visual.Window* property), 423
- ColorCAL (class in *psychopy.hardware.crs.colorcal*), 504
- colorSpace () (*psychopy.visual.BoxStim* property), 160
- colorSpace () (*psychopy.visual.BufferImageStim* property), 169
- colorSpace () (*psychopy.visual.circle.Circle* property), 179

- colorSpace () (*psychopy.visual.Form* property), 194
- colorSpace () (*psychopy.visual.GratingStim* property), 203
- colorSpace () (*psychopy.visual.ImageStim* property), 213
- colorSpace () (*psychopy.visual.line.Line* property), 224
- colorSpace () (*psychopy.visual.MovieStim* property), 233
- colorSpace () (*psychopy.visual.nnlvs.VisualSystemHD* property), 405
- colorSpace () (*psychopy.visual.ObjMeshStim* property), 246
- colorSpace () (*psychopy.visual.pie.Pie* property), 257
- colorSpace () (*psychopy.visual.PlaneStim* property), 265
- colorSpace () (*psychopy.visual.polygon.Polygon* property), 274
- colorSpace () (*psychopy.visual.RadialStim* property), 283
- colorSpace () (*psychopy.visual.rect.Rect* property), 297
- colorSpace () (*psychopy.visual.rift.Rift* property), 311
- colorSpace () (*psychopy.visual.shape.ShapeStim* property), 345
- colorSpace () (*psychopy.visual.SphereStim* property), 358
- colorSpace () (*psychopy.visual.TextBox2* property), 374
- colorSpace () (*psychopy.visual.TextStim* property), 383
- colorSpace () (*psychopy.visual.Window* property), 424
- compileShader () (in module *psychopy.tools.gltools*), 591
- compileShaderObjectARB () (in module *psychopy.tools.gltools*), 591
- complete () (*psychopy.clock.StaticPeriod* method), 154
- complete () (*psychopy.core.StaticPeriod* method), 151
- complete () (*psychopy.visual.Form* property), 195
- compress () (*psychopy.microphone.AdvAudioCapture* method), 757
- computeBoundingBoxCorners () (in module *psychopy.tools.mathtools*), 670
- computeChecksum () (in module *psychopy.plugins*), 771
- computeFrustum () (in module *psychopy.tools.viewtools*), 683
- computeFrustumFOV () (in module *psychopy.tools.viewtools*), 684
- concatenate () (in module *psychopy.tools.mathtools*), 664
- confidence_interval (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent* attribute), 550
- confidence_interval (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 549
- confInterval () (*psychopy.data.QuestHandler* method), 723
- connectedControllers () (*psychopy.visual.rift.Rift* property), 311
- contains () (*psychopy.visual.Aperture* method), 156
- contains () (*psychopy.visual.BufferImageStim* method), 169
- contains () (*psychopy.visual.circle.Circle* method), 179
- contains () (*psychopy.visual.Form* method), 195
- contains () (*psychopy.visual.GratingStim* method), 203
- contains () (*psychopy.visual.ImageStim* method), 214
- contains () (*psychopy.visual.line.Line* method), 225
- contains () (*psychopy.visual.MovieStim* method), 233
- contains () (*psychopy.visual.pie.Pie* method), 258
- contains () (*psychopy.visual.polygon.Polygon* method), 274
- contains () (*psychopy.visual.RadialStim* method), 284
- contains () (*psychopy.visual.rect.Rect* method), 298
- contains () (*psychopy.visual.shape.ShapeStim* method), 346
- contains () (*psychopy.visual.TextBox2* method), 375
- contains () (*psychopy.visual.TextStim* method), 383
- contains () (*psychopy.visual.VlcMovieStim* method), 392
- contentScaleFactor () (*psychopy.visual.nnlvs.VisualSystemHD* property), 405
- contentScaleFactor () (*psychopy.visual.rift.Rift* property), 311
- contentScaleFactor () (*psychopy.visual.Window* property), 424
- contrast (*psychopy.visual.Slider* attribute), 353
- contrast () (*psychopy.visual.BoxStim* property), 161
- contrast () (*psychopy.visual.BufferImageStim* property), 169
- contrast () (*psychopy.visual.circle.Circle* property), 180
- contrast () (*psychopy.visual.Form* property), 195
- contrast () (*psychopy.visual.GratingStim* property), 204
- contrast () (*psychopy.visual.ImageStim* property), 214
- contrast () (*psychopy.visual.line.Line* property), 225
- contrast () (*psychopy.visual.MovieStim* property), 234

- `contrast ()` (*psychopy.visual.ObjMeshStim* property), 246
`contrast ()` (*psychopy.visual.pie.Pie* property), 258
`contrast ()` (*psychopy.visual.PlaneStim* property), 266
`contrast ()` (*psychopy.visual.polygon.Polygon* property), 274
`contrast ()` (*psychopy.visual.RadialStim* property), 284
`contrast ()` (*psychopy.visual.rect.Rect* property), 298
`contrast ()` (*psychopy.visual.shape.ShapeStim* property), 346
`contrast ()` (*psychopy.visual.SphereStim* property), 358
`contrast ()` (*psychopy.visual.TextBox2* property), 375
`contrast ()` (*psychopy.visual.TextStim* property), 383
`conv2d ()` (in module *psychopy.visual.filters*), 742
`convergeOffset ()` (*psychopy.visual.nnlvs.VisualSystemHD* property), 405
`convergeOffset ()` (*psychopy.visual.rift.Rift* property), 311
`convergeOffset ()` (*psychopy.visual.Window* property), 424
`convertToPix ()` (in module *psychopy.tools.monitorunittools*), 678
`convertToWAV ()` (*psychopy.sound.AudioClip* method), 454
`coordToRay ()` (*psychopy.visual.nnlvs.VisualSystemHD* method), 406
`coordToRay ()` (*psychopy.visual.rift.Rift* method), 312
`coordToRay ()` (*psychopy.visual.Window* method), 424
`copy ()` (*psychopy.colors.Color* method), 694
`copy ()` (*psychopy.layout.Position* method), 750
`copy ()` (*psychopy.layout.Size* method), 752
`copy ()` (*psychopy.layout.Vector* method), 749
`copy ()` (*psychopy.sound.AudioClip* method), 455
`copy ()` (*psychopy.visual.RigidBodyPose* method), 336
`copyCalib ()` (*psychopy.monitors.Monitor* method), 762
`CountdownTimer` (class in *psychopy.clock*), 152
`CountdownTimer` (class in *psychopy.core*), 149
CPU, 30
`cpuLoad ()` (*psychopy.sound.AudioDeviceStatus* property), 463
`createBoundingBox ()` (*psychopy.visual.rift.Rift* static method), 313
`createBox ()` (in module *psychopy.tools.gltools*), 629
`createCubeMap ()` (in module *psychopy.tools.gltools*), 608
`createFBO ()` (in module *psychopy.tools.gltools*), 600
`createFromPTBDesc ()` (*psychopy.sound.AudioDeviceInfo* static method), 460
`createFromPTBDesc ()` (*psychopy.sound.AudioDeviceStatus* static method), 463
`createHapticsBuffer ()` (*psychopy.visual.rift.Rift* static method), 313
`createLight ()` (in module *psychopy.tools.gltools*), 621
`createMaterial ()` (in module *psychopy.tools.gltools*), 619
`createMeshGrid ()` (in module *psychopy.tools.gltools*), 628
`createMeshGridFromArrays ()` (in module *psychopy.tools.gltools*), 627
`createPlane ()` (in module *psychopy.tools.gltools*), 626
`createPose ()` (*psychopy.visual.rift.Rift* static method), 314
`createProgram ()` (in module *psychopy.tools.gltools*), 589
`createProgramObjectARB ()` (in module *psychopy.tools.gltools*), 590
`createQueryObject ()` (in module *psychopy.tools.gltools*), 598
`createRenderbuffer ()` (in module *psychopy.tools.gltools*), 604
`createTexImage2D ()` (in module *psychopy.tools.gltools*), 605
`createTexImage2dFromFile ()` (in module *psychopy.tools.gltools*), 606
`createTexImage2DMultisample ()` (in module *psychopy.tools.gltools*), 607
`createTrialHandlerRecordTable ()` (*psychopy.iohub.client.ioHubConnection* method), 527
`createUVSphere ()` (in module *psychopy.tools.gltools*), 625
`createVAO ()` (in module *psychopy.tools.gltools*), 610
`createVBO ()` (in module *psychopy.tools.gltools*), 613
`critical ()` (in module *psychopy.logging*), 755
`cross ()` (in module *psychopy.tools.mathtools*), 638
CRT, 30
CS100A (class in *psychopy.hardware.minolta*), 515
csv, 30
`cullFace ()` (*psychopy.visual.nnlvs.VisualSystemHD* property), 407
`cullFace ()` (*psychopy.visual.rift.Rift* property), 314
`cullFace ()` (*psychopy.visual.Window* property), 425
`cullFaceMode ()` (*psychopy.visual.nnlvs.VisualSystemHD* property), 407
`cullFaceMode ()` (*psychopy.visual.rift.Rift* property), 314
`cullFaceMode ()` (*psychopy.visual.Window* property), 425

- 425
 currentEditable () (psy-
 choppy.visual.nnlvs.VisualSystemHD property),
 407
 currentEditable () (psychopy.visual.rift.Rift prop-
 erty), 314
 currentEditable () (psychopy.visual.Window prop-
 erty), 425
 currentLoop () (psychopy.data.ExperimentHandler
 property), 697
 currentStreamTime () (psy-
 choppy.sound.AudioDeviceStatus property),
 463
 cursorToRay () (in module psychopy.tools.viewtools),
 690
 CustomMouse (class in psychopy.visual), 185
- ## D
- daemon () (psychopy.hardware.emulator.ResponseEmulator
 property), 505
 daemon () (psychopy.hardware.emulator.SyncGenerator
 property), 507
 data () (in module psychopy.logging), 755
 data () (psychopy.data.TrialHandler2 property), 705
 dcReset () (psychopy.hardware.brainproducts.RemoteControlServer
 method), 468
 debug () (in module psychopy.logging), 755
 decreaseVolume () (psychopy.visual.VlcMovieStim
 method), 392
 defaultSampleRate () (psy-
 choppy.sound.AudioDeviceInfo property),
 461
 deg () (psychopy.layout.Position property), 750
 deg () (psychopy.layout.Size property), 752
 deg () (psychopy.layout.Vector property), 749
 deg () (psychopy.layout.Vertices property), 753
 deg2cm () (in module psychopy.tools.monitorunittools),
 678
 deg2pix () (in module psy-
 choppy.tools.monitorunittools), 678
 degFlat () (psychopy.layout.Position property), 750
 degFlat () (psychopy.layout.Size property), 752
 degFlat () (psychopy.layout.Vector property), 749
 degFlat () (psychopy.layout.Vertices property), 753
 degFlatPos () (psychopy.layout.Position property),
 751
 degFlatPos () (psychopy.layout.Size property), 752
 degFlatPos () (psychopy.layout.Vector property), 749
 degrees () (in module psychopy.tools.unittools), 681
 delay (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent
 attribute), 550
 delay (psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent
 attribute), 549
 delCalib () (psychopy.monitors.Monitor method), 762
 deleteCaretLeft () (psychopy.visual.TextBox2
 method), 375
 deleteCaretRight () (psychopy.visual.TextBox2
 method), 375
 deleteFBO () (in module psychopy.tools.gltools), 602
 deleteObject () (in module psychopy.tools.gltools),
 593
 deleteObjectARB () (in module psy-
 choppy.tools.gltools), 593
 deleteRenderbuffer () (in module psy-
 choppy.tools.gltools), 604
 deleteTexture () (in module psy-
 choppy.tools.gltools), 607
 deleteVAO () (in module psychopy.tools.gltools), 612
 deleteVBO () (in module psychopy.tools.gltools), 616
 delta () (psychopy.data.QuestHandler property), 723
 depth (psychopy.visual.BufferImageStim attribute), 170
 depth (psychopy.visual.circle.Circle attribute), 180
 depth (psychopy.visual.Form attribute), 195
 depth (psychopy.visual.GratingStim attribute), 204
 depth (psychopy.visual.ImageStim attribute), 215
 depth (psychopy.visual.line.Line attribute), 225
 depth (psychopy.visual.MovieStim attribute), 234
 depth (psychopy.visual.pie.Pie attribute), 258
 depth (psychopy.visual.polygon.Polygon attribute), 275
 depth (psychopy.visual.RadialStim attribute), 284
 depth (psychopy.visual.rect.Rect attribute), 298
 depth (psychopy.visual.shape.ShapeStim attribute), 346
 depth (psychopy.visual.TextBox2 attribute), 375
 depth (psychopy.visual.TextStim attribute), 384
 depth (psychopy.visual.VlcMovieStim attribute), 393
 depthFunc () (psychopy.visual.nnlvs.VisualSystemHD
 property), 407
 depthFunc () (psychopy.visual.rift.Rift property), 314
 depthFunc () (psychopy.visual.Window property), 425
 depthMask () (psychopy.visual.nnlvs.VisualSystemHD
 property), 407
 depthMask () (psychopy.visual.rift.Rift property), 314
 depthMask () (psychopy.visual.Window property), 425
 depthTest () (psychopy.visual.nnlvs.VisualSystemHD
 property), 407
 depthTest () (psychopy.visual.rift.Rift property), 314
 depthTest () (psychopy.visual.Window property), 425
 detachObjectARB () (in module psy-
 choppy.tools.gltools), 594
 detachShader () (in module psychopy.tools.gltools),
 594
 detect () (psychopy.voicekey.OnsetVoiceKey method),
 774
 detect () (psychopy.iohub.client.keyboard.KeyboardPress
 property), 533
 detect () (psychopy.iohub.client.keyboard.KeyboardRelease
 property), 534

device_time (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent* attribute), 550

device_time (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 549

deviceIndex () (*psychopy.sound.AudioDeviceInfo* property), 461

deviceName () (*psychopy.sound.AudioDeviceInfo* property), 461

diffuseColor () (*psychopy.visual.BlinnPhongMaterial* property), 253

diffuseColor () (*psychopy.visual.LightSource* property), 221

diffuseRGB () (*psychopy.visual.BlinnPhongMaterial* property), 253

diffuseRGB () (*psychopy.visual.LightSource* property), 221

diffuseTexture () (*psychopy.visual.BlinnPhongMaterial* property), 253

dimensions () (*psychopy.layout.Position* property), 751

dimensions () (*psychopy.layout.Size* property), 752

dimensions () (*psychopy.layout.Vector* property), 749

diopeters () (*psychopy.visual.nnlvs.VisualSystemHD* property), 407

dir (*psychopy.visual.DotStim* attribute), 187

direction () (*psychopy.layout.Position* property), 751

direction () (*psychopy.layout.Size* property), 752

direction () (*psychopy.layout.Vector* property), 750

disable () (*psychopy.visual.Aperture* method), 156

disableVertexArray () (in module *psychopy.tools.gltools*), 619

dispatchAllWindowsEvents () (*psychopy.visual.nnlvs.VisualSystemHD* class method), 407

dispatchAllWindowsEvents () (*psychopy.visual.rift.Rift* class method), 314

dispense () (*psychopy.hardware.qmix.Pump* method), 521

displayRefreshRate () (*psychopy.visual.rift.Rift* property), 314

displayResolution () (*psychopy.visual.rift.Rift* property), 314

distance () (in module *psychopy.tools.mathtools*), 640

distanceTo () (*psychopy.visual.RigidBodyPose* method), 336

distCoef () (*psychopy.visual.nnlvs.VisualSystemHD* property), 407

dkl () (*psychopy.colors.Color* property), 694

dkl2rgb () (in module *psychopy.tools.colorspectools*), 583

dkla () (*psychopy.colors.Color* property), 694

dklaCart () (*psychopy.colors.Color* property), 694

dklCart2rgb () (in module *psychopy.colors.colorspectools*), 583

Dlg (class in *psychopy.gui*), 745

DlgFromDict (class in *psychopy.gui*), 744

dot () (in module *psychopy.tools.mathtools*), 637

dotLife (*psychopy.visual.DotStim* attribute), 186

dotSize (*psychopy.visual.DotStim* attribute), 186

DotStim (class in *psychopy.visual*), 186

draw () (*psychopy.visual.BoxStim* method), 161

draw () (*psychopy.visual.BufferImageStim* method), 170

draw () (*psychopy.visual.circle.Circle* method), 180

draw () (*psychopy.visual.Form* method), 195

draw () (*psychopy.visual.GratingStim* method), 204

draw () (*psychopy.visual.ImageStim* method), 215

draw () (*psychopy.visual.line.Line* method), 225

draw () (*psychopy.visual.MovieStim* method), 234

draw () (*psychopy.visual.ObjMeshStim* method), 247

draw () (*psychopy.visual.pie.Pie* method), 259

draw () (*psychopy.visual.PlaneStim* method), 266

draw () (*psychopy.visual.polygon.Polygon* method), 275

draw () (*psychopy.visual.RadialStim* method), 284

draw () (*psychopy.visual.rect.Rect* method), 298

draw () (*psychopy.visual.SceneSkybox* method), 340

draw () (*psychopy.visual.shape.ShapeStim* method), 346

draw () (*psychopy.visual.Slider* method), 353

draw () (*psychopy.visual.SphereStim* method), 358

draw () (*psychopy.visual.TextBox* method), 367

draw () (*psychopy.visual.TextBox2* method), 375

draw () (*psychopy.visual.TextStim* method), 384

draw () (*psychopy.visual.VlcMovieStim* method), 393

draw3d () (*psychopy.visual.nnlvs.VisualSystemHD* property), 407

draw3d () (*psychopy.visual.rift.Rift* property), 314

draw3d () (*psychopy.visual.Window* property), 426

drawVAO () (in module *psychopy.tools.gltools*), 611

driverFor (*psychopy.hardware.crs.bits.BitsSharp* attribute), 485

driverFor (*psychopy.hardware.crs.colorcal.ColorCAL* attribute), 504

duration (*psychopy.iohub.devices.eyetracker.BlinkEndEvent* attribute), 561

duration (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 540, 558, 579

duration (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 560

duration () (*psychopy.iohub.client.keyboard.KeyboardRelease* property), 534

duration () (*psychopy.sound.AudioClip* property), 455

duration () (*psychopy.visual.MovieStim* property), 234

duration () (*psychopy.visual.VlcMovieStim* property), 393

E

- edges (*psychopy.visual.circle.Circle* attribute), 180
- edges (*psychopy.visual.polygon.Polygon* attribute), 275
- editable () (*psychopy.visual.TextBox2* property), 376
- elapsedOutSamples () (*psychopy.sound.AudioDeviceStatus* property), 463
- element (*psychopy.visual.DotStim* attribute), 186
- ElementArrayStim (class in *psychopy.visual*), 188
- embedShaderSourceDefs () (in module *psychopy.tools.gltools*), 592
- emissionColor () (*psychopy.visual.BlinnPhongMaterial* property), 253
- emissionRGB () (*psychopy.visual.BlinnPhongMaterial* property), 253
- empty () (*psychopy.hardware.qmix.Pump* method), 522
- enable () (*psychopy.visual.Aperture* method), 156
- enabled (*psychopy.visual.Aperture* attribute), 156
- enableEventReporting () (*psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker* method), 537
- enableEventReporting () (*psychopy.iohub.devices.eyetracker.hw.mouse.EyeTracker* method), 576
- enableEventReporting () (*psychopy.iohub.devices.eyetracker.hw.tobii.EyeTracker* method), 568
- enableVertexArray () (in module *psychopy.tools.gltools*), 618
- end (*psychopy.visual.line.Line* attribute), 226
- end (*psychopy.visual.pie.Pie* attribute), 259
- end () (*psychopy.visual.BlinnPhongMaterial* method), 253
- end_angle_x (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 558
- end_angle_x (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 560
- end_angle_y (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 558
- end_angle_y (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 560
- end_gaze_x (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 558, 579
- end_gaze_x (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 560
- end_gaze_y (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 558, 579
- end_gaze_y (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 560
- end_ppd_x (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 558
- end_ppd_x (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 560
- end_ppd_y (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 558
- end_ppd_y (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 560
- end_pupil_measure1_type (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 558
- end_pupil_measure1_type (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 560
- end_pupil_measure_1 (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 558
- end_pupil_measure_1 (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 560
- end_velocity_xy (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 558
- end_velocity_xy (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 561
- endOffFlip () (*psychopy.visual.windowframepack.ProjectorFramePacker* method), 436
- endQuery () (in module *psychopy.tools.gltools*), 599
- endRemoteMode () (*psychopy.hardware.pr.PR655* method), 519
- enforceWASAPI (*psychopy.sound.Microphone* attribute), 450
- EnvelopeGrating (class in *psychopy.visual*), 341
- epsilon () (*psychopy.data.QuestHandler* property), 723
- error () (in module *psychopy.logging*), 755
- estimatedStopTime () (*psychopy.sound.AudioDeviceStatus* property), 463
- estimateLambda () (*psychopy.data.PsiHandler* method), 718
- estimateThreshold () (*psychopy.data.PsiHandler* method), 719
- eval () (*psychopy.data.FitCumNormal* method), 736
- eval () (*psychopy.data.FitLogistic* method), 735
- eval () (*psychopy.data.FitNakaRushton* method), 735
- eval () (*psychopy.data.FitWeibull* method), 734
- exp () (in module *psychopy.logging*), 755
- ExperimentHandler (class in *psychopy.data*), 696
- expName () (*psychopy.hardware.brainproducts.RemoteControlServer* property), 468
- extent () (*psychopy.visual.Slider* property), 353
- extents () (*psychopy.visual.BoundingBox* property), 158
- eye (*psychopy.iohub.devices.eyetracker.BlinkEndEvent* attribute), 561

eye (*psychopy.iohub.devices.eyetracker.BlinkStartEvent* attribute), 561
 eye (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 540, 557, 579
 eye (*psychopy.iohub.devices.eyetracker.FixationStartEvent* attribute), 540, 557, 578
 eye (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 549, 554
 eye (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 560
 eye (*psychopy.iohub.devices.eyetracker.SaccadeStartEvent* attribute), 559
 eye_cam_x (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 550
 eye_cam_y (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 550
 eye_cam_z (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 550
 eyeHeight () (*psychopy.visual.rift.Rift* property), 315
 eyeOffset () (*psychopy.visual.nnlvs.VisualSystemHD* property), 407
 eyeOffset () (*psychopy.visual.rift.Rift* property), 315
 eyeOffset () (*psychopy.visual.Window* property), 426
 eyeRenderPose () (*psychopy.visual.rift.Rift* property), 315
 eyeToNoseDistance () (*psychopy.visual.rift.Rift* property), 315
 EyeTracker (class in *psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3*), 536
 EyeTracker (class in *psychopy.iohub.devices.eyetracker.hw.mouse*), 575
 EyeTracker (class in *psychopy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core*), 546
 EyeTracker (class in *psychopy.iohub.devices.eyetracker.hw.tobii*), 567
 F
 fadeOut () (*psychopy.sound.backend_pygame.SoundPygame* method), 447
 farClip () (*psychopy.visual.nnlvs.VisualSystemHD* property), 407
 farClip () (*psychopy.visual.rift.Rift* property), 315
 farClip () (*psychopy.visual.Window* property), 426
 fastForward () (*psychopy.visual.MovieStim* method), 234
 fastForward () (*psychopy.visual.VlcMovieStim* method), 393
 fatal () (in module *psychopy.logging*), 755
 fieldPos (*psychopy.visual.DotStim* attribute), 187
 fieldShape (*psychopy.visual.DotStim* attribute), 186
 fieldSize (*psychopy.visual.DotStim* attribute), 187
 filename () (*psychopy.visual.MovieStim* property), 235
 filename () (*psychopy.visual.VlcMovieStim* property), 393
 fileOpenDlg () (in module *psychopy.gui*), 746
 fileSaveDlg () (in module *psychopy.gui*), 746
 fill () (*psychopy.hardware.qmix.Pump* method), 522
 fillColor () (*psychopy.visual.BoxStim* property), 161
 fillColor () (*psychopy.visual.BufferImageStim* property), 170
 fillColor () (*psychopy.visual.circle.Circle* property), 180
 fillColor () (*psychopy.visual.Form* property), 195
 fillColor () (*psychopy.visual.GratingStim* property), 204
 fillColor () (*psychopy.visual.ImageStim* property), 215
 fillColor () (*psychopy.visual.line.Line* property), 226
 fillColor () (*psychopy.visual.MovieStim* property), 235
 fillColor () (*psychopy.visual.ObjMeshStim* property), 247
 fillColor () (*psychopy.visual.pie.Pie* property), 259
 fillColor () (*psychopy.visual.PlaneStim* property), 266
 fillColor () (*psychopy.visual.polygon.Polygon* property), 275
 fillColor () (*psychopy.visual.RadialStim* property), 284
 fillColor () (*psychopy.visual.rect.Rect* property), 298
 fillColor () (*psychopy.visual.shape.ShapeStim* property), 346
 fillColor () (*psychopy.visual.Slider* property), 353
 fillColor () (*psychopy.visual.SphereStim* property), 358
 fillColor () (*psychopy.visual.TextBox2* property), 376
 fillColorSpace () (*psychopy.visual.BoxStim* property), 161
 fillColorSpace () (*psychopy.visual.BufferImageStim* property), 170
 fillColorSpace () (*psychopy.visual.circle.Circle* property), 180
 fillColorSpace () (*psychopy.visual.Form* property), 196
 fillColorSpace () (*psychopy.visual.GratingStim* property), 204
 fillColorSpace () (*psychopy.visual.ImageStim* property), 215
 fillColorSpace () (*psychopy.visual.line.Line* prop-

- erty*), 226
- `fillColorSpace()` (*psychopy.visual.MovieStim property*), 235
- `fillColorSpace()` (*psychopy.visual.ObjMeshStim property*), 247
- `fillColorSpace()` (*psychopy.visual.pie.Pie property*), 259
- `fillColorSpace()` (*psychopy.visual.PlaneStim property*), 266
- `fillColorSpace()` (*psychopy.visual.polygon.Polygon property*), 275
- `fillColorSpace()` (*psychopy.visual.RadialStim property*), 284
- `fillColorSpace()` (*psychopy.visual.rect.Rect property*), 298
- `fillColorSpace()` (*psychopy.visual.shape.ShapeStim property*), 346
- `fillColorSpace()` (*psychopy.visual.SphereStim property*), 358
- `fillColorSpace()` (*psychopy.visual.TextBox2 property*), 376
- `fillLevel()` (*psychopy.hardware.qmix.Pump property*), 522
- `fillRGB()` (*psychopy.visual.BoxStim property*), 161
- `fillRGB()` (*psychopy.visual.BufferImageStim property*), 170
- `fillRGB()` (*psychopy.visual.circle.Circle property*), 180
- `fillRGB()` (*psychopy.visual.Form property*), 196
- `fillRGB()` (*psychopy.visual.GratingStim property*), 204
- `fillRGB()` (*psychopy.visual.ImageStim property*), 215
- `fillRGB()` (*psychopy.visual.line.Line property*), 226
- `fillRGB()` (*psychopy.visual.MovieStim property*), 235
- `fillRGB()` (*psychopy.visual.ObjMeshStim property*), 247
- `fillRGB()` (*psychopy.visual.pie.Pie property*), 259
- `fillRGB()` (*psychopy.visual.PlaneStim property*), 266
- `fillRGB()` (*psychopy.visual.polygon.Polygon property*), 275
- `fillRGB()` (*psychopy.visual.RadialStim property*), 284
- `fillRGB()` (*psychopy.visual.rect.Rect property*), 298
- `fillRGB()` (*psychopy.visual.shape.ShapeStim property*), 346
- `fillRGB()` (*psychopy.visual.SphereStim property*), 359
- `fillRGB()` (*psychopy.visual.TextBox2 property*), 376
- `findPhotometer()` (*in module psychopy.hardware*), 522
- `firmwareVersion()` (*psychopy.visual.rift.Rift property*), 315
- `fit()` (*psychopy.visual.BoundingBox method*), 158
- `fitBBBox()` (*in module psychopy.tools.mathtools*), 670
- `FitCumNormal` (*class in psychopy.data*), 736
- `fitGammaErrFun()` (*psychopy.monitors.GammaCalculator method*), 765
- `fitGammaFun()` (*psychopy.monitors.GammaCalculator method*), 765
- `FitLogistic` (*class in psychopy.data*), 735
- `FitNakaRushton` (*class in psychopy.data*), 735
- `FitWeibull` (*class in psychopy.data*), 734
- `FixationEndEvent` (*class in psychopy.iohub.devices.eyetracker*), 540, 557, 578
- `FixationStartEvent` (*class in psychopy.iohub.devices.eyetracker*), 540, 557, 578
- `fixTangentHandedness()` (*in module psychopy.tools.mathtools*), 645
- `flac2wav()` (*in module psychopy.microphone*), 759
- `flip()` (*psychopy.layout.Vertices property*), 753
- `flip()` (*psychopy.visual.Aperture property*), 156
- `flip()` (*psychopy.visual.BoxStim property*), 161
- `flip()` (*psychopy.visual.BufferImageStim property*), 170
- `flip()` (*psychopy.visual.circle.Circle property*), 180
- `flip()` (*psychopy.visual.Form property*), 196
- `flip()` (*psychopy.visual.GratingStim property*), 204
- `flip()` (*psychopy.visual.ImageStim property*), 215
- `flip()` (*psychopy.visual.line.Line property*), 226
- `flip()` (*psychopy.visual.MovieStim property*), 235
- `flip()` (*psychopy.visual.nnlvs.VisualSystemHD method*), 407
- `flip()` (*psychopy.visual.ObjMeshStim property*), 247
- `flip()` (*psychopy.visual.pie.Pie property*), 259
- `flip()` (*psychopy.visual.PlaneStim property*), 266
- `flip()` (*psychopy.visual.polygon.Polygon property*), 275
- `flip()` (*psychopy.visual.RadialStim property*), 285
- `flip()` (*psychopy.visual.rect.Rect property*), 299
- `flip()` (*psychopy.visual.rift.Rift method*), 315
- `flip()` (*psychopy.visual.shape.ShapeStim property*), 347
- `flip()` (*psychopy.visual.Slider property*), 353
- `flip()` (*psychopy.visual.SphereStim property*), 359
- `flip()` (*psychopy.visual.TextBox2 property*), 376
- `flip()` (*psychopy.visual.TextStim property*), 384
- `flip()` (*psychopy.visual.VlcMovieStim property*), 393
- `flip()` (*psychopy.visual.Window method*), 426
- `flipHoriz` (*psychopy.visual.BufferImageStim attribute*), 170
- `flipHoriz` (*psychopy.visual.TextStim attribute*), 384
- `flipHoriz()` (*psychopy.layout.Vertices property*), 753
- `flipHoriz()` (*psychopy.visual.BoxStim property*), 161
- `flipHoriz()` (*psychopy.visual.Form property*), 196

- flipHoriz () (*psychopy.visual.GratingStim* property), 204
- flipHoriz () (*psychopy.visual.ImageStim* property), 215
- flipHoriz () (*psychopy.visual.MovieStim* property), 235
- flipHoriz () (*psychopy.visual.ObjMeshStim* property), 247
- flipHoriz () (*psychopy.visual.PlaneStim* property), 266
- flipHoriz () (*psychopy.visual.RadialStim* property), 285
- flipHoriz () (*psychopy.visual.SphereStim* property), 359
- flipHoriz () (*psychopy.visual.TextBox2* property), 376
- flipHoriz () (*psychopy.visual.VlcMovieStim* property), 393
- flipVert (*psychopy.visual.BufferImageStim* attribute), 170
- flipVert (*psychopy.visual.TextStim* attribute), 384
- flipVert () (*psychopy.layout.Vertices* property), 753
- flipVert () (*psychopy.visual.BoxStim* property), 161
- flipVert () (*psychopy.visual.Form* property), 196
- flipVert () (*psychopy.visual.GratingStim* property), 204
- flipVert () (*psychopy.visual.ImageStim* property), 215
- flipVert () (*psychopy.visual.MovieStim* property), 235
- flipVert () (*psychopy.visual.ObjMeshStim* property), 247
- flipVert () (*psychopy.visual.PlaneStim* property), 266
- flipVert () (*psychopy.visual.RadialStim* property), 285
- flipVert () (*psychopy.visual.SphereStim* property), 359
- flipVert () (*psychopy.visual.TextBox2* property), 376
- flipVert () (*psychopy.visual.VlcMovieStim* property), 393
- float_uint16 () (in module *psychopy.tools.typetools*), 680
- float_uint8 () (in module *psychopy.tools.typetools*), 680
- flowRateUnit () (*psychopy.hardware.qmix.Pump* property), 522
- flush () (in module *psychopy.logging*), 755, 756
- flush () (*psychopy.hardware.crs.bits.BitsSharp* method), 485
- flush () (*psychopy.logging._Logger* method), 754
- flush () (*psychopy.sound.Microphone* method), 450
- flushDataStoreFile () (*psychopy.iohub.client.ioHubConnection* method), 528
- font (*psychopy.visual.TextBox2* attribute), 376
- font (*psychopy.visual.TextStim* attribute), 384
- fontFiles (*psychopy.visual.TextStim* attribute), 384
- fontMGR () (*psychopy.visual.TextBox2* property), 376
- foreColor () (*psychopy.visual.BoxStim* property), 161
- foreColor () (*psychopy.visual.BufferImageStim* property), 170
- foreColor () (*psychopy.visual.circle.Circle* property), 180
- foreColor () (*psychopy.visual.Form* property), 196
- foreColor () (*psychopy.visual.GratingStim* property), 204
- foreColor () (*psychopy.visual.ImageStim* property), 215
- foreColor () (*psychopy.visual.line.Line* property), 226
- foreColor () (*psychopy.visual.MovieStim* property), 235
- foreColor () (*psychopy.visual.ObjMeshStim* property), 247
- foreColor () (*psychopy.visual.pie.Pie* property), 259
- foreColor () (*psychopy.visual.PlaneStim* property), 266
- foreColor () (*psychopy.visual.polygon.Polygon* property), 275
- foreColor () (*psychopy.visual.RadialStim* property), 285
- foreColor () (*psychopy.visual.rect.Rect* property), 299
- foreColor () (*psychopy.visual.shape.ShapeStim* property), 347
- foreColor () (*psychopy.visual.Slider* property), 353
- foreColor () (*psychopy.visual.SphereStim* property), 359
- foreColor () (*psychopy.visual.TextBox2* property), 376
- foreColor () (*psychopy.visual.TextStim* property), 384
- foreColorSpace () (*psychopy.visual.BoxStim* property), 162
- foreColorSpace () (*psychopy.visual.BufferImageStim* property), 171
- foreColorSpace () (*psychopy.visual.circle.Circle* property), 181
- foreColorSpace () (*psychopy.visual.Form* property), 196
- foreColorSpace () (*psychopy.visual.GratingStim* property), 205
- foreColorSpace () (*psychopy.visual.ImageStim* property), 216
- foreColorSpace () (*psychopy.visual.line.Line* property), 227
- foreColorSpace () (*psychopy.visual.MovieStim* property), 235

- property*), 236
 - foreColorSpace () (*psychopy.visual.ObjMeshStim property*), 248
 - foreColorSpace () (*psychopy.visual.pie.Pie property*), 260
 - foreColorSpace () (*psychopy.visual.PlaneStim property*), 267
 - foreColorSpace () (*psychopy.visual.polygon.Polygon property*), 276
 - foreColorSpace () (*psychopy.visual.RadialStim property*), 285
 - foreColorSpace () (*psychopy.visual.rect.Rect property*), 299
 - foreColorSpace () (*psychopy.visual.shape.ShapeStim property*), 347
 - foreColorSpace () (*psychopy.visual.SphereStim property*), 360
 - foreColorSpace () (*psychopy.visual.TextBox2 property*), 377
 - foreColorSpace () (*psychopy.visual.TextStim property*), 385
 - foreRGB () (*psychopy.visual.BoxStim property*), 162
 - foreRGB () (*psychopy.visual.BufferImageStim property*), 171
 - foreRGB () (*psychopy.visual.circle.Circle property*), 181
 - foreRGB () (*psychopy.visual.Form property*), 196
 - foreRGB () (*psychopy.visual.GratingStim property*), 205
 - foreRGB () (*psychopy.visual.ImageStim property*), 216
 - foreRGB () (*psychopy.visual.line.Line property*), 227
 - foreRGB () (*psychopy.visual.MovieStim property*), 236
 - foreRGB () (*psychopy.visual.ObjMeshStim property*), 248
 - foreRGB () (*psychopy.visual.pie.Pie property*), 260
 - foreRGB () (*psychopy.visual.PlaneStim property*), 267
 - foreRGB () (*psychopy.visual.polygon.Polygon property*), 276
 - foreRGB () (*psychopy.visual.RadialStim property*), 286
 - foreRGB () (*psychopy.visual.rect.Rect property*), 299
 - foreRGB () (*psychopy.visual.shape.ShapeStim property*), 347
 - foreRGB () (*psychopy.visual.SphereStim property*), 360
 - foreRGB () (*psychopy.visual.TextBox2 property*), 377
 - foreRGB () (*psychopy.visual.TextStim property*), 385
 - Form (*class in psychopy.visual*), 191
 - formComplete () (*psychopy.visual.Form method*), 196
 - forwardProject () (*in module psychopy.tools.mathtools*), 666
 - fps () (*psychopy.visual.MovieStim property*), 236
 - fps () (*psychopy.visual.nnlvs.VisualSystemHD method*), 408
 - fps () (*psychopy.visual.rift.Rift method*), 316
 - fps () (*psychopy.visual.VlcMovieStim property*), 393
 - fps () (*psychopy.visual.Window method*), 426
 - frameBufferSize () (*psychopy.visual.nnlvs.VisualSystemHD property*), 408
 - frameBufferSize () (*psychopy.visual.rift.Rift property*), 316
 - frameBufferSize () (*psychopy.visual.Window property*), 426
 - frameIndex () (*psychopy.visual.MovieStim property*), 236
 - frameIndex () (*psychopy.visual.VlcMovieStim property*), 393
 - frameRate () (*psychopy.visual.MovieStim property*), 236
 - frameSize () (*psychopy.visual.MovieStim property*), 236
 - frameTexture () (*psychopy.visual.MovieStim property*), 236
 - frameTexture () (*psychopy.visual.VlcMovieStim property*), 393
 - frameTime () (*psychopy.visual.VlcMovieStim property*), 394
 - fromFile () (*in module psychopy.tools.filetools*), 587
 - frontFace () (*psychopy.visual.nnlvs.VisualSystemHD property*), 408
 - frontFace () (*psychopy.visual.rift.Rift property*), 316
 - frontFace () (*psychopy.visual.Window property*), 426
 - fullscr (*psychopy.visual.nnlvs.VisualSystemHD attribute*), 408
 - fullscr (*psychopy.visual.rift.Rift attribute*), 316
 - fullscr (*psychopy.visual.Window attribute*), 426
 - functionFromStaircase () (*in module psychopy.data*), 737
- ## G
- gain () (*psychopy.sound.AudioClip method*), 455
 - gamma (*psychopy.visual.nnlvs.VisualSystemHD attribute*), 408
 - gamma (*psychopy.visual.rift.Rift attribute*), 316
 - gamma (*psychopy.visual.Window attribute*), 426
 - gamma () (*psychopy.data.QuestHandler property*), 723
 - GammaCalculator (*class in psychopy.monitors*), 765
 - gammaCorrectFile () (*psychopy.hardware.crs.bits.BitsSharp property*), 485
 - gammaFun () (*in module psychopy.monitors*), 766
 - gammaInvFun () (*in module psychopy.monitors*), 766
 - gammaIsDefault () (*psychopy.monitors.Monitor method*), 762
 - gammaRamp (*psychopy.visual.nnlvs.VisualSystemHD attribute*), 408

gammaRamp (*psychopy.visual.rift.Rift* attribute), 316
 gammaRamp (*psychopy.visual.Window* attribute), 427
 gaze_x (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent* attribute), 578
 gaze_x (*psychopy.iohub.devices.eyetracker.FixationStartEvent* attribute), 540, 557, 578
 gaze_x (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 549, 554
 gaze_x (*psychopy.iohub.devices.eyetracker.SaccadeStartEvent* attribute), 559
 gaze_y (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent* attribute), 578
 gaze_y (*psychopy.iohub.devices.eyetracker.FixationStartEvent* attribute), 540, 557, 578
 gaze_y (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 549, 554
 gaze_y (*psychopy.iohub.devices.eyetracker.SaccadeStartEvent* attribute), 559
 gaze_z (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 549
 genDelimiter () (in module *psychopy.tools.filetools*), 588
 generalizedPerspectiveProjection () (in module *psychopy.tools.viewtools*), 686
 get_a () (*psychopy.hardware.joystick.XboxController* method), 512
 get_b () (*psychopy.hardware.joystick.XboxController* method), 512
 get_back () (*psychopy.hardware.joystick.XboxController* method), 512
 get_hat_axis () (*psychopy.hardware.joystick.XboxController* method), 512
 get_left_shoulder () (*psychopy.hardware.joystick.XboxController* method), 512
 get_left_thumbstick () (*psychopy.hardware.joystick.XboxController* method), 512
 get_left_thumbstick_axis () (*psychopy.hardware.joystick.XboxController* method), 512
 get_named_buttons () (*psychopy.hardware.joystick.XboxController* method), 512
 get_right_shoulder () (*psychopy.hardware.joystick.XboxController* method), 512
 get_right_thumbstick () (*psychopy.hardware.joystick.XboxController* method), 513
 get_right_thumbstick_axis () (*psychopy.hardware.joystick.XboxController* method), 513
 get_start () (*psychopy.hardware.joystick.XboxController* method), 513
 get_trigger_axis () (*psychopy.hardware.joystick.XboxController* method), 513
 get_x () (*psychopy.hardware.joystick.XboxController* method), 513
 get_y () (*psychopy.hardware.joystick.XboxController* method), 513
 getAbsTime () (in module *psychopy.clock*), 154
 getAbsTime () (in module *psychopy.core*), 151
 getAbsTimeGPU () (in module *psychopy.tools.gltools*), 599
 getActualFrameRate () (*psychopy.visual.nnlvs.VisualSystemHD* method), 408
 getActualFrameRate () (*psychopy.visual.rift.Rift* method), 316
 getActualFrameRate () (*psychopy.visual.Window* method), 427
 getAllAxes () (*psychopy.hardware.joystick.Joystick* method), 513
 getAllButtons () (*psychopy.hardware.joystick.Joystick* method), 513
 getAllEntries () (*psychopy.data.ExperimentHandler* method), 697
 getAllHats () (*psychopy.hardware.joystick.Joystick* method), 513
 getAllMonitors () (in module *psychopy.monitors*), 765
 getAllRTBoxResponses () (*psychopy.hardware.crs.bits.BitsSharp* method), 485
 getAllStatusBoxResponses () (*psychopy.hardware.crs.bits.BitsSharp* method), 485
 getAllStatusEvents () (*psychopy.hardware.crs.bits.BitsSharp* method), 486
 getAllStatusValues () (*psychopy.hardware.crs.bits.BitsSharp* method), 486
 getAnalog () (*psychopy.hardware.crs.bits.BitsSharp* method), 487
 getAppFrame () (in module *psychopy.app*), 693
 getAppInstance () (in module *psychopy.app*), 693
 getas () (*psychopy.layout.Vertices* method), 753
 getAttribLocations () (in module *psychopy.tools.gltools*), 597
 getAutoLog () (*psychopy.visual.TextBox* method), 367
 getAxis () (*psychopy.hardware.joystick.Joystick* method), 513

getBackend() (*psychopy.hardware.keyboard.Keyboard class method*), 465
 getBackgroundColor() (*psychopy.visual.TextBox method*), 367
 getBorderColor() (*psychopy.visual.TextBox method*), 367
 getBorderWidth() (*psychopy.visual.TextBox method*), 367
 getBoundaryDimensions() (*psychopy.visual.rift.Rift method*), 316
 getButton() (*psychopy.hardware.joystick.Joystick method*), 513
 getButtons() (*psychopy.visual.rift.Rift method*), 317
 getCalibDate() (*psychopy.monitors.Monitor method*), 762
 getCalibMatrix() (*psychopy.hardware.crs.colorcal.ColorCAL method*), 504
 getColorSpace() (*psychopy.visual.TextBox method*), 367
 getConfiguration() (*psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker method*), 537
 getConfiguration() (*psychopy.iohub.devices.eyetracker.hw.mouse.EyeTracker method*), 576
 getConfiguration() (*psychopy.iohub.devices.eyetracker.hw.tobii.EyeTracker method*), 568
 getContentScaleFactor() (*psychopy.visual.nnlvs.VisualSystemHD method*), 409
 getContentScaleFactor() (*psychopy.visual.rift.Rift method*), 317
 getContentScaleFactor() (*psychopy.visual.Window method*), 427
 getCurrentFrameNumber() (*psychopy.visual.MovieStim method*), 236
 getCurrentFrameNumber() (*psychopy.visual.VlcMovieStim method*), 394
 getCurrentFrameTime() (*psychopy.visual.VlcMovieStim method*), 394
 getCurrentTrial() (*psychopy.data.TrialHandler method*), 700
 getCurrentTrial() (*psychopy.data.TrialHandlerExt method*), 710
 getCurrentTrialPosInDataHandler() (*psychopy.data.TrialHandlerExt method*), 710
 getData() (*psychopy.visual.Form method*), 196
 getDevice() (*psychopy.iohub.client.ioHubConnection method*), 525
 getDevicePose() (*psychopy.visual.rift.Rift method*), 318
 getDevices() (*psychopy.sound.Microphone static method*), 450
 getDeviceSN() (*psychopy.hardware.pr.PR655 method*), 519
 getDeviceType() (*psychopy.hardware.pr.PR655 method*), 519
 getDft() (*in module psychopy.microphone*), 759
 getDigital() (*psychopy.hardware.crs.bits.BitsSharp method*), 487
 getDigitalWord() (*psychopy.hardware.crs.bits.BitsSharp method*), 487
 getDisplayedText() (*psychopy.visual.TextBox method*), 367
 getDistance() (*psychopy.monitors.Monitor method*), 762
 getDKL_RGB() (*psychopy.monitors.Monitor method*), 762
 getDuration() (*psychopy.sound.backend_pygame.SoundPygame method*), 447
 getEarlierTrial() (*psychopy.data.TrialHandler method*), 700
 getEarlierTrial() (*psychopy.data.TrialHandler2 method*), 705
 getEarlierTrial() (*psychopy.data.TrialHandlerExt method*), 710
 getEvents() (*psychopy.hardware.forp.ButtonBox method*), 510
 getEvents() (*psychopy.iohub.client.ioHubConnection method*), 526
 getEvents() (*psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker method*), 537
 getEvents() (*psychopy.iohub.devices.eyetracker.hw.mouse.EyeTracker method*), 576
 getEvents() (*psychopy.iohub.devices.eyetracker.hw.tobii.EyeTracker method*), 568
 getExp() (*psychopy.data.MultiStairHandler method*), 732
 getExp() (*psychopy.data.PsiHandler method*), 719
 getExp() (*psychopy.data.QuestHandler method*), 723
 getExp() (*psychopy.data.QuestPlusHandler method*), 728
 getExp() (*psychopy.data.StairHandler method*), 715
 getExp() (*psychopy.data.TrialHandler method*), 700
 getExp() (*psychopy.data.TrialHandler2 method*), 705
 getExp() (*psychopy.data.TrialHandlerExt method*), 710
 getFloatv() (*in module psychopy.tools.gltools*), 632
 getFontColor() (*psychopy.visual.TextBox method*), 367
 getFontSize() (*psychopy.visual.TextBox method*), 368
 getFPS() (*psychopy.visual.MovieStim method*), 236

- getFPS () (*psychopy.visual.VlcMovieStim method*), 394
- getFutureFlipTime () (*psychopy.visual.nnlvs.VisualSystemHD method*), 409
- getFutureFlipTime () (*psychopy.visual.rift.Rift method*), 318
- getFutureFlipTime () (*psychopy.visual.Window method*), 428
- getFutureTrial () (*psychopy.data.TrialHandler method*), 700
- getFutureTrial () (*psychopy.data.TrialHandler2 method*), 705
- getFutureTrial () (*psychopy.data.TrialHandlerExt method*), 710
- getGamma () (*psychopy.monitors.Monitor method*), 762
- getGammaGrid () (*psychopy.monitors.Monitor method*), 762
- getGlyphPositionForTextIndex () (*psychopy.visual.TextBox method*), 368
- getHandTriggerValues () (*psychopy.visual.rift.Rift method*), 318
- getHat () (*psychopy.hardware.joystick.Joystick method*), 514
- getHistory () (*psychopy.visual.Slider method*), 354
- getHorzAlign () (*psychopy.visual.TextBox method*), 368
- getHorzJust () (*psychopy.visual.TextBox method*), 368
- getIndexTriggerValues () (*psychopy.visual.rift.Rift method*), 319
- getInfo () (*psychopy.hardware.crs.bits.BitsSharp method*), 488
- getInfo () (*psychopy.hardware.crs.colorcal.ColorCAL method*), 504
- getInfoLog () (*in module psychopy.tools.gltools*), 596
- getIntegerv () (*in module psychopy.tools.gltools*), 632
- getInterpolated () (*psychopy.visual.TextBox method*), 368
- getIRBox () (*psychopy.hardware.crs.bits.BitsSharp method*), 488
- getKeyboards () (*in module psychopy.hardware.keyboard*), 466
- getKeys () (*in module psychopy.event*), 740
- getKeys () (*psychopy.hardware.keyboard.Keyboard method*), 465
- getKeys () (*psychopy.iohub.client.keyboard.Keyboard method*), 531
- getLabel () (*psychopy.visual.TextBox method*), 368
- getLastColorTemp () (*psychopy.hardware.pr.PR655 method*), 519
- getLastGazePosition () (*psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker method*), 537
- getLastGazePosition () (*psychopy.iohub.devices.eyetracker.hw.mouse.EyeTracker method*), 576
- getLastGazePosition () (*psychopy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core.EyeTracker method*), 547
- getLastGazePosition () (*psychopy.iohub.devices.eyetracker.hw.tobii.EyeTracker method*), 569
- getLastLum () (*psychopy.hardware.pr.PR650 method*), 518
- getLastResetTime () (*psychopy.clock.MonotonicClock method*), 153
- getLastResetTime () (*psychopy.core.MonotonicClock method*), 150
- getLastSample () (*psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker method*), 538
- getLastSample () (*psychopy.iohub.devices.eyetracker.hw.mouse.EyeTracker method*), 577
- getLastSample () (*psychopy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core.EyeTracker method*), 547
- getLastSample () (*psychopy.iohub.devices.eyetracker.hw.tobii.EyeTracker method*), 569
- getLastSpectrum () (*psychopy.hardware.pr.PR650 method*), 518
- getLastSpectrum () (*psychopy.hardware.pr.PR655 method*), 519
- getLastTristim () (*psychopy.hardware.pr.PR655 method*), 519
- getLastUV () (*psychopy.hardware.pr.PR655 method*), 520
- getLastXY () (*psychopy.hardware.pr.PR655 method*), 520
- getLevel () (*in module psychopy.logging*), 755
- getLevelsPost () (*psychopy.monitors.Monitor method*), 762
- getLevelsPre () (*psychopy.monitors.Monitor method*), 762
- getLinearizeMethod () (*psychopy.monitors.Monitor method*), 763
- getLineSpacing () (*psychopy.visual.TextBox method*), 368
- getLMS_RGB () (*psychopy.monitors.Monitor method*), 762
- getLoudness () (*psychopy.microphone.AdvAudioCapture method*), 757
- getLum () (*psychopy.hardware.crs.colorcal.ColorCAL method*), 504
- getLum () (*psychopy.hardware.minolta.CS100A method*), 504

- method*), 516
- getLum() (*psychopy.hardware.minolta.LS100 method*), 517
- getLum() (*psychopy.hardware.pr.PR650 method*), 518
- getLumSeriesPR650() (*in module psychopy.monitors*), 765
- getLumsPost() (*psychopy.monitors.Monitor method*), 763
- getLumsPre() (*psychopy.monitors.Monitor method*), 763
- getMarkerInfo() (*psychopy.microphone.AdvAudioCapture method*), 757
- getMarkerOnset() (*psychopy.microphone.AdvAudioCapture method*), 757
- getMarkerPos() (*psychopy.visual.Slider method*), 354
- getMeanLum() (*psychopy.monitors.Monitor method*), 763
- getMemoryUsage() (*in module psychopy.info*), 748
- getModelMatrix() (*psychopy.visual.RigidBodyPose method*), 337
- getModelViewMatrix() (*in module psychopy.tools.gltools*), 633
- getMouseResponses() (*psychopy.visual.Slider method*), 354
- getMovieFrame() (*psychopy.visual.nnlvs.VisualSystemHD method*), 409
- getMovieFrame() (*psychopy.visual.rift.Rift method*), 319
- getMovieFrame() (*psychopy.visual.Window method*), 428
- getMsPerFrame() (*psychopy.visual.nnlvs.VisualSystemHD method*), 410
- getMsPerFrame() (*psychopy.visual.rift.Rift method*), 319
- getMsPerFrame() (*psychopy.visual.Window method*), 428
- getName() (*psychopy.hardware.joystick.Joystick method*), 514
- getName() (*psychopy.visual.TextBox method*), 368
- getNeedsCalibrateZero() (*psychopy.hardware.crs.colorcal.ColorCAL method*), 504
- getNextTrialPosInDataHandler() (*psychopy.data.TrialHandlerExt method*), 710
- getNormalMatrix() (*psychopy.visual.RigidBodyPose method*), 337
- getNotes() (*psychopy.monitors.Monitor method*), 763
- getNumAxes() (*psychopy.hardware.joystick.Joystick method*), 514
- getNumButtons() (*psychopy.hardware.joystick.Joystick method*), 514
- getNumHats() (*psychopy.hardware.joystick.Joystick method*), 514
- getNumJoysticks() (*in module psychopy.hardware.joystick*), 513
- getOpacity() (*psychopy.visual.TextBox method*), 368
- getOpenGLInfo() (*in module psychopy.tools.gltools*), 632
- getOri() (*psychopy.visual.BoxStim method*), 162
- getOri() (*psychopy.visual.ObjMeshStim method*), 248
- getOri() (*psychopy.visual.PlaneStim method*), 267
- getOri() (*psychopy.visual.SphereStim method*), 360
- getOriAxisAngle() (*psychopy.visual.BoxStim method*), 162
- getOriAxisAngle() (*psychopy.visual.ObjMeshStim method*), 248
- getOriAxisAngle() (*psychopy.visual.PlaneStim method*), 267
- getOriAxisAngle() (*psychopy.visual.RigidBodyPose method*), 337
- getOriAxisAngle() (*psychopy.visual.SphereStim method*), 360
- getOriginPathAndFile() (*psychopy.data.MultiStairHandler method*), 732
- getOriginPathAndFile() (*psychopy.data.PsiHandler method*), 719
- getOriginPathAndFile() (*psychopy.data.QuestHandler method*), 723
- getOriginPathAndFile() (*psychopy.data.QuestPlusHandler method*), 728
- getOriginPathAndFile() (*psychopy.data.StairHandler method*), 715
- getOriginPathAndFile() (*psychopy.data.TrialHandler method*), 701
- getOriginPathAndFile() (*psychopy.data.TrialHandler2 method*), 705
- getOriginPathAndFile() (*psychopy.data.TrialHandlerExt method*), 710
- getPackets() (*psychopy.hardware.crs.bits.BitsPlusPlus method*), 473
- getPackets() (*psychopy.hardware.crs.bits.BitsSharp method*), 488
- getPercentageComplete() (*psychopy.visual.MovieStim method*), 236
- getPercentageComplete() (*psychopy.visual.VlcMovieStim method*), 394
- getPos() (*psychopy.event.Mouse method*), 738
- getPos() (*psychopy.visual.BoxStim method*), 163
- getPos() (*psychopy.visual.ObjMeshStim method*), 248

getPos () (*psychopy.visual.PlaneStim* method), 267
 getPos () (*psychopy.visual.SphereStim* method), 360
 getPosition () (psy-
 chopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker
 method), 538
 getPosition () (psy-
 chopy.iohub.devices.eyetracker.hw.mouse.EyeTracker
 method), 577
 getPosition () (psy-
 chopy.iohub.devices.eyetracker.hw.tobii.EyeTracker
 method), 570
 getPosition () (*psychopy.visual.TextBox* method),
 368
 getPredictedDisplayTime () (psy-
 chopy.visual.rift.Rift method), 320
 getPressed () (*psychopy.event.Mouse* method), 738
 getPresses () (psy-
 chopy.iohub.client.keyboard.Keyboard
 method), 531
 getPriority () (psy-
 chopy.iohub.client.ioHubConnection method),
 528
 getProcessAffinity () (psy-
 chopy.iohub.client.ioHubConnection method),
 528
 getProjectionMatrix () (in module psy-
 chopy.tools.gltools), 633
 getPsychopyVersion () (psy-
 chopy.monitors.Monitor method), 763
 getQuery () (in module *psychopy.tools.gltools*), 599
 getRAM () (in module *psychopy.info*), 748
 getRating () (*psychopy.visual.Slider* method), 354
 getRayIntersectBounds () (psy-
 chopy.visual.BoxStim method), 163
 getRayIntersectBounds () (psy-
 chopy.visual.ObjMeshStim method), 248
 getRayIntersectBounds () (psy-
 chopy.visual.PlaneStim method), 268
 getRayIntersectBounds () (psy-
 chopy.visual.SphereStim method), 360
 getRayIntersectSphere () (psy-
 chopy.visual.SphereStim method), 360
 getRecording () (*psychopy.sound.Microphone*
 method), 450
 getRel () (*psychopy.event.Mouse* method), 738
 getReleases () (psy-
 chopy.iohub.client.keyboard.Keyboard
 method), 531
 getResponse () (psy-
 chopy.hardware.crs.bits.BitsSharp method),
 489
 getRGBspectra () (in module *psychopy.monitors*),
 766
 getRMS () (in module *psychopy.microphone*), 759
 getRMScontrast () (in module psy-
 chopy.visual.filters), 742
 getRT () (*psychopy.visual.Slider* method), 354
 EyeTrackerResponse () (psy-
 chopy.hardware.crs.bits.BitsSharp method),
 488
 getRTBoxResponses () (psy-
 chopy.hardware.crs.bits.BitsSharp method),
 488
 getSize () (*psychopy.visual.TextBox* method), 368
 getSizePix () (*psychopy.monitors.Monitor* method),
 763
 getSpectra () (*psychopy.monitors.Monitor* method),
 763
 getSpectrum () (*psychopy.hardware.pr.PR650*
 method), 518
 getStatus () (*psychopy.hardware.crs.bits.BitsSharp*
 method), 489
 getStatusBoxResponse () (psy-
 chopy.hardware.crs.bits.BitsSharp method),
 490
 getStatusBoxResponses () (psy-
 chopy.hardware.crs.bits.BitsSharp method),
 490
 getStatusEvent () (psy-
 chopy.hardware.crs.bits.BitsSharp method),
 490
 getString () (in module *psychopy.tools.gltools*), 632
 getText () (*psychopy.visual.TextBox* method), 368
 getText () (*psychopy.visual.TextBox2* method), 377
 getTextGridCellForCharIndex () (psy-
 chopy.visual.TextBox method), 368
 getTextGridCellPlacement () (psy-
 chopy.visual.TextBox method), 368
 getTextGridLineColor () (psy-
 chopy.visual.TextBox method), 369
 getTextGridLineWidth () (psy-
 chopy.visual.TextBox method), 369
 getThumbstickValues () (*psychopy.visual.rift.Rift*
 method), 320
 getTime () (in module *psychopy.clock*), 154
 getTime () (in module *psychopy.core*), 151
 getTime () (*psychopy.clock.CountdownTimer* method),
 153
 getTime () (*psychopy.clock.MonotonicClock* method),
 153
 getTime () (*psychopy.core.CountdownTimer* method),
 150
 getTime () (*psychopy.core.MonotonicClock* method),
 150
 getTime () (*psychopy.iohub.client.ioHubConnection*
 method), 528
 getTimeInSeconds () (*psychopy.visual.rift.Rift*
 method), 320

[getTouches\(\)](#) (*psychopy.visual.rift.Rift method*), 320
[getTrackerInfo\(\)](#) (*psychopy.visual.rift.Rift method*), 321
[getTrackingState\(\)](#) (*psychopy.visual.rift.Rift method*), 321
[getTrigIn\(\)](#) (*psychopy.hardware.crs.bits.BitsSharp method*), 491
[getUniformLocations\(\)](#) (*in module psychopy.tools.gltools*), 597
[getUniqueEvents\(\)](#) (*psychopy.hardware.forp.ButtonBox method*), 511
[getUnitType\(\)](#) (*psychopy.visual.TextBox method*), 369
[getUseBits\(\)](#) (*psychopy.monitors.Monitor method*), 763
[getValidStrokeWidths\(\)](#) (*psychopy.visual.TextBox method*), 369
[getVertAlign\(\)](#) (*psychopy.visual.TextBox method*), 369
[getVertJust\(\)](#) (*psychopy.visual.TextBox method*), 369
[getVideoLine\(\)](#) (*psychopy.hardware.crs.bits.BitsSharp method*), 491
[getViewMatrix\(\)](#) (*psychopy.visual.RigidBodyPose method*), 338
[getVisible\(\)](#) (*psychopy.event.Mouse method*), 738
[getVisibleText\(\)](#) (*psychopy.visual.TextBox2 method*), 377
[getVolume\(\)](#) (*psychopy.sound.backend_pygame.SoundPygame method*), 448
[getVolume\(\)](#) (*psychopy.visual.VlcMovieStim method*), 394
[getWheelRel\(\)](#) (*psychopy.event.Mouse method*), 738
[getWidth\(\)](#) (*psychopy.monitors.Monitor method*), 763
[getWindow\(\)](#) (*psychopy.visual.TextBox method*), 369
[getX\(\)](#) (*psychopy.hardware.joystick.Joystick method*), 514
[getY\(\)](#) (*psychopy.hardware.joystick.Joystick method*), 514
[getYawPitchRoll\(\)](#) (*psychopy.visual.RigidBodyPose method*), 338
[getZ\(\)](#) (*psychopy.hardware.joystick.Joystick method*), 514
GPU, 30
[grain\(\)](#) (*psychopy.data.QuestHandler property*), 723
[GratingStim](#) (*class in psychopy.visual*), 201
[groupFlipVert\(\)](#) (*in module psychopy.visual.helpers*), 210

H

[hasFocus\(\)](#) (*psychopy.visual.TextBox2 property*), 377
[hasInputFocus\(\)](#) (*psychopy.visual.rift.Rift property*), 323
[hasMagYawCorrection\(\)](#) (*psychopy.visual.rift.Rift property*), 323
[hasOrientationTracking\(\)](#) (*psychopy.visual.rift.Rift property*), 323
[hasPositionTracking\(\)](#) (*psychopy.visual.rift.Rift property*), 323
[haveInternetAccess\(\)](#) (*in module psychopy.web*), 776
[headLocked\(\)](#) (*psychopy.visual.rift.Rift property*), 323
[height](#) (*psychopy.visual.TextStim attribute*), 385
[height\(\)](#) (*psychopy.layout.Position property*), 751
[height\(\)](#) (*psychopy.layout.Size property*), 752
[height\(\)](#) (*psychopy.layout.Vector property*), 750
[height\(\)](#) (*psychopy.layout.Vertices property*), 753
[height\(\)](#) (*psychopy.visual.BoxStim property*), 163
[height\(\)](#) (*psychopy.visual.BufferImageStim property*), 171
[height\(\)](#) (*psychopy.visual.Form property*), 196
[height\(\)](#) (*psychopy.visual.GratingStim property*), 205
[height\(\)](#) (*psychopy.visual.ImageStim property*), 216
[height\(\)](#) (*psychopy.visual.MovieStim property*), 236
[height\(\)](#) (*psychopy.visual.ObjMeshStim property*), 249
[height\(\)](#) (*psychopy.visual.PlaneStim property*), 268
[height\(\)](#) (*psychopy.visual.RadialStim property*), 286
[height\(\)](#) (*psychopy.visual.SphereStim property*), 360
[height\(\)](#) (*psychopy.visual.TextBox2 property*), 377
[height\(\)](#) (*psychopy.visual.VlcMovieStim property*), 394
[hex\(\)](#) (*psychopy.colors.Color property*), 694
[hex2rgb255\(\)](#) (*in module psychopy.colors*), 695
[hid\(\)](#) (*psychopy.visual.rift.Rift property*), 323
[hidePerfHud\(\)](#) (*psychopy.visual.rift.Rift method*), 323
[hmdMounted\(\)](#) (*psychopy.visual.rift.Rift property*), 323
[hmdPresent\(\)](#) (*psychopy.visual.rift.Rift property*), 323
[horiz\(\)](#) (*psychopy.visual.Slider property*), 354
[hostAPIName\(\)](#) (*psychopy.sound.AudioDeviceInfo property*), 461
[hsv\(\)](#) (*psychopy.colors.Color property*), 694
[hsv2rgb\(\)](#) (*in module psychopy.tools.colors spacetools*), 584
[hsva\(\)](#) (*psychopy.colors.Color property*), 695

I

[ident\(\)](#) (*psychopy.hardware.emulator.ResponseEmulator property*), 505
[ident\(\)](#) (*psychopy.hardware.emulator.SyncGenerator property*), 507

- image (*psychopy.visual.BufferImageStim* attribute), 171
- image (*psychopy.visual.ImageStim* attribute), 216
- image2array () (in module *psychopy.tools.imagetools*), 634
- ImageStim (class in *psychopy.visual*), 212
- imfft () (in module *psychopy.visual.filters*), 742
- imifft () (in module *psychopy.visual.filters*), 742
- importConditions () (in module *psychopy.data*), 736
- importData () (*psychopy.data.QuestHandler* method), 723
- importItems () (*psychopy.visual.Form* method), 196
- increaseVolume () (*psychopy.visual.VlcMovieStim* method), 394
- incTrials () (*psychopy.data.QuestHandler* method), 724
- inDeviceIndex () (*psychopy.sound.AudioDeviceStatus* property), 463
- info () (in module *psychopy.logging*), 755
- inputChannels () (*psychopy.sound.AudioDeviceInfo* property), 461
- inputLatency () (*psychopy.sound.AudioDeviceInfo* property), 461
- intensity () (*psychopy.data.MultiStairHandler* property), 732
- intensity () (*psychopy.data.PsiHandler* property), 719
- intensity () (*psychopy.data.QuestHandler* property), 724
- intensity () (*psychopy.data.QuestPlusHandler* property), 728
- intensity () (*psychopy.data.StairHandler* property), 715
- interp () (*psychopy.visual.RigidBodyPose* method), 338
- interpolate (*psychopy.visual.BufferImageStim* attribute), 171
- interpolate (*psychopy.visual.circle.Circle* attribute), 181
- interpolate (*psychopy.visual.GratingStim* attribute), 205
- interpolate (*psychopy.visual.ImageStim* attribute), 216
- interpolate (*psychopy.visual.line.Line* attribute), 227
- interpolate (*psychopy.visual.pie.Pie* attribute), 260
- interpolate (*psychopy.visual.polygon.Polygon* attribute), 276
- interpolate (*psychopy.visual.RadialStim* attribute), 286
- interpolate (*psychopy.visual.rect.Rect* attribute), 300
- interpolate (*psychopy.visual.shape.ShapeStim* attribute), 348
- interpolate () (*psychopy.visual.VlcMovieStim* property), 394
- intersectRayAABB () (in module *psychopy.tools.mathtools*), 672
- intersectRayOBB () (in module *psychopy.tools.mathtools*), 673
- intersectRayPlane () (in module *psychopy.tools.mathtools*), 671
- intersectRaySphere () (in module *psychopy.tools.mathtools*), 671
- intersectRayTriangle () (in module *psychopy.tools.mathtools*), 674
- inverse () (*psychopy.data.FitCumNormal* method), 736
- inverse () (*psychopy.data.FitLogistic* method), 735
- inverse () (*psychopy.data.FitNakaRushton* method), 735
- inverse () (*psychopy.data.FitWeibull* method), 735
- inverseModelMatrix () (*psychopy.visual.RigidBodyPose* property), 338
- invert () (*psychopy.visual.Aperture* method), 156
- invert () (*psychopy.visual.RigidBodyPose* method), 338
- inverted (*psychopy.visual.Aperture* attribute), 156
- inverted () (*psychopy.visual.RigidBodyPose* method), 338
- invertMatrix () (in module *psychopy.tools.mathtools*), 662
- invertQuat () (in module *psychopy.tools.mathtools*), 656
- ioHubConnection (class in *psychopy.iohub.client*), 525
- is_alive () (*psychopy.hardware.emulator.ResponseEmulator* method), 506
- is_alive () (*psychopy.hardware.emulator.SyncGenerator* method), 507
- isAffine () (in module *psychopy.tools.mathtools*), 663
- isAlive () (*psychopy.hardware.emulator.ResponseEmulator* method), 506
- isAlive () (*psychopy.hardware.emulator.SyncGenerator* method), 507
- isAppStarted () (in module *psychopy.app*), 693
- isAwake () (*psychopy.hardware.crs.bits.BitsSharp* method), 491
- isBoundaryVisible () (*psychopy.visual.rift.Rift* property), 323
- isCapture () (*psychopy.sound.AudioDeviceInfo* property), 461
- isCapture () (*psychopy.sound.AudioDeviceStatus* property), 463
- isComplete () (in module *psychopy.tools.gltools*), 602
- isConnected () (*psychopy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core.EyeTrac*

- method*), 547
 - `isDuplex()` (*psychopy.sound.AudioDeviceInfo* property), 461
 - `isDuplex()` (*psychopy.sound.AudioDeviceStatus* property), 463
 - `isEqual()` (*psychopy.visual.RigidBodyPose* method), 338
 - `isFinished()` (*psychopy.visual.MovieStim* property), 236
 - `isFinished()` (*psychopy.visual.VlcMovieStim* property), 395
 - `isHmdConnected()` (in module *psychopy.tools.rifttools*), 679
 - `isInFaultState()` (*psychopy.hardware.qmix.Pump* property), 522
 - `isMono()` (*psychopy.sound.AudioClip* property), 455
 - `isNotStarted()` (*psychopy.visual.MovieStim* property), 236
 - `isNotStarted()` (*psychopy.visual.VlcMovieStim* property), 395
 - `isOculusServiceRunning()` (in module *psychopy.tools.rifttools*), 679
 - `isOpen()` (*psychopy.hardware.crs.bits.BitsSharp* property), 492
 - `isOrthogonal()` (in module *psychopy.tools.mathtools*), 662
 - `isPaused()` (*psychopy.visual.MovieStim* property), 237
 - `isPaused()` (*psychopy.visual.VlcMovieStim* property), 395
 - `isPlayback()` (*psychopy.sound.AudioDeviceInfo* property), 461
 - `isPlayback()` (*psychopy.sound.AudioDeviceStatus* property), 463
 - `isPlaying()` (*psychopy.visual.MovieStim* property), 237
 - `isPlaying()` (*psychopy.visual.VlcMovieStim* property), 395
 - `isPoseVisible()` (*psychopy.visual.rift.Rift* method), 323
 - `isPressedIn()` (*psychopy.event.Mouse* method), 739
 - `isRecBufferFull()` (*psychopy.sound.Microphone* property), 450
 - `isRecordingEnabled()` (*psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker* property), 538
 - `isRecordingEnabled()` (*psychopy.iohub.devices.eyetracker.hw.mouse.EyeTracker* property), 577
 - `isRecordingEnabled()` (*psychopy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core.EyeTracker* property), 547
 - `isRecordingEnabled()` (*psychopy.iohub.devices.eyetracker.hw.tobii.EyeTracker* property), 570
 - `isStarted()` (*psychopy.sound.Microphone* property), 450
 - `isStereo()` (*psychopy.sound.AudioClip* property), 455
 - `isStopped()` (*psychopy.visual.MovieStim* property), 237
 - `isStopped()` (*psychopy.visual.VlcMovieStim* property), 395
 - `isValid()` (*psychopy.visual.BoundingBox* property), 158
 - `isValidColor()` (in module *psychopy.colors*), 695
 - `isVisible()` (*psychopy.visual.BoxStim* method), 163
 - `isVisible()` (*psychopy.visual.ObjMeshStim* method), 249
 - `isVisible()` (*psychopy.visual.PlaneStim* method), 268
 - `isVisible()` (*psychopy.visual.rift.Rift* property), 324
 - `isVisible()` (*psychopy.visual.SphereStim* method), 360
 - italic* (*psychopy.visual.TextStim* attribute), 385
 - `itemColor()` (*psychopy.visual.Form* property), 196
- ## J
- `join()` (*psychopy.hardware.emulator.ResponseEmulator* method), 506
 - `join()` (*psychopy.hardware.emulator.SyncGenerator* method), 507
 - `join()` (*psychopy.voicekey.OnsetVoiceKey* method), 774
 - Joystick* (class in *psychopy.hardware.joystick*), 513
- ## K
- Keyboard* (class in *psychopy.hardware.keyboard*), 465
 - Keyboard* (class in *psychopy.iohub.client.keyboard*), 530
 - KeyPress* (class in *psychopy.iohub.client.keyboard*), 533
 - KeyboardRelease* (class in *psychopy.iohub.client.keyboard*), 534
 - KeyPress* (class in *psychopy.hardware.keyboard*), 466
 - knownStyles* (*psychopy.visual.Form* attribute), 196
 - knownStyles* (*psychopy.visual.Slider* attribute), 355
 - knownStyleTweaks* (*psychopy.visual.Slider* attribute), 354
- ## L
- `labelColor()` (*psychopy.visual.Slider* property), 355
 - `labelHeight()` (*psychopy.visual.Slider* property), 355
 - `labelWidth()` (*psychopy.visual.Slider* property), 355
 - `languageStyle()` (*psychopy.visual.TextBox2* property), 377

latencyBias() (*psychopy.sound.AudioDeviceStatus property*), 463

latencyBias() (*psychopy.sound.Microphone property*), 450

launchHubServer() (*in module psychopy.iohub.client*), 524

launchScan() (*in module psychopy.hardware.emulator*), 508

left_angle_x (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 551, 555

left_angle_y (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 551, 555

left_eye_cam_x (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 551, 571

left_eye_cam_y (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 551, 571

left_eye_cam_z (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 551, 571

left_gaze_x (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 539, 550, 555, 570

left_gaze_y (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 539, 550, 555, 570

left_gaze_z (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 551

left_ppd_x (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 555

left_ppd_y (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 556

left_pupil_measure1 (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 551

left_pupil_measure1_type (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 551, 555

left_pupil_measure2 (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 551

left_pupil_measure_1 (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 539, 555, 571, 578

left_raw_x (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 539, 551, 555

left_raw_y (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 539, 551, 555

left_velocity_x (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 556

left_velocity_xy (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 556

attribute), 556

left_velocity_y (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 556

legacyStyles (*psychopy.visual.Slider attribute*), 355

legacyStyleTweaks (*psychopy.visual.Slider attribute*), 355

length() (*in module psychopy.tools.mathtools*), 635

lensCorrection() (*in module psychopy.tools.mathtools*), 674

lensCorrection() (*psychopy.visual.nnlvs.VisualSystemHD property*), 410

lensCorrectionSpherical() (*in module psychopy.tools.mathtools*), 675

letterHeight() (*psychopy.visual.TextBox2 property*), 377

letterHeightPix() (*psychopy.visual.TextBox2 property*), 377

LibOVRBounds (*in module psychopy.tools.rifttools*), 679

LibOVRHapticsBuffer (*in module psychopy.tools.rifttools*), 679

LibOVRPose (*in module psychopy.tools.rifttools*), 679

LibOVRState (*in module psychopy.tools.rifttools*), 679

libOVRVisualSystemHD (*psychopy.visual.nnlvs.VisualSystemHD property*), 410

libOVRWindow (*psychopy.visual.rift.Rift property*), 324

lights() (*psychopy.visual.Window property*), 429

LightSource (*class in psychopy.visual*), 220

lightType() (*psychopy.visual.LightSource property*), 221

line (*class in psychopy.visual.line*), 222

lineariseLums() (*psychopy.monitors.Monitor method*), 763

normalizeLums() (*psychopy.monitors.Monitor method*), 763

lineColor() (*psychopy.visual.BoxStim property*), 163

lineColor() (*psychopy.visual.BufferImageStim property*), 171

lineColor() (*psychopy.visual.circle.Circle property*), 181

lineColor() (*psychopy.visual.Form property*), 196

lineColor() (*psychopy.visual.GratingStim property*), 206

lineColor() (*psychopy.visual.ImageStim property*), 216

lineColor() (*psychopy.visual.line.Line property*), 227

lineColor() (*psychopy.visual.MovieStim property*), 237

lineColor() (*psychopy.visual.ObjMeshStim property*), 247

- `erty`), 249
- `lineColor()` (*psychopy.visual.pie.Pie* property), 260
- `lineColor()` (*psychopy.visual.PlaneStim* property), 268
- `lineColor()` (*psychopy.visual.polygon.Polygon* property), 276
- `lineColor()` (*psychopy.visual.RadialStim* property), 286
- `lineColor()` (*psychopy.visual.rect.Rect* property), 300
- `lineColor()` (*psychopy.visual.shape.ShapeStim* property), 348
- `lineColor()` (*psychopy.visual.SphereStim* property), 361
- `lineColor()` (*psychopy.visual.TextBox2* property), 377
- `lineColorSpace()` (*psychopy.visual.BoxStim* property), 163
- `lineColorSpace()` (*psychopy.visual.BufferImageStim* property), 172
- `lineColorSpace()` (*psychopy.visual.circle.Circle* property), 181
- `lineColorSpace()` (*psychopy.visual.Form* property), 196
- `lineColorSpace()` (*psychopy.visual.GratingStim* property), 206
- `lineColorSpace()` (*psychopy.visual.ImageStim* property), 216
- `lineColorSpace()` (*psychopy.visual.line.Line* property), 227
- `lineColorSpace()` (*psychopy.visual.MovieStim* property), 237
- `lineColorSpace()` (*psychopy.visual.ObjMeshStim* property), 249
- `lineColorSpace()` (*psychopy.visual.pie.Pie* property), 260
- `lineColorSpace()` (*psychopy.visual.PlaneStim* property), 268
- `lineColorSpace()` (*psychopy.visual.polygon.Polygon* property), 276
- `lineColorSpace()` (*psychopy.visual.RadialStim* property), 286
- `lineColorSpace()` (*psychopy.visual.rect.Rect* property), 300
- `lineColorSpace()` (*psychopy.visual.shape.ShapeStim* property), 348
- `lineColorSpace()` (*psychopy.visual.SphereStim* property), 361
- `lineColorSpace()` (*psychopy.visual.TextBox2* property), 377
- `lineRGB()` (*psychopy.visual.BoxStim* property), 163
- `lineRGB()` (*psychopy.visual.BufferImageStim* property), 172
- `lineRGB()` (*psychopy.visual.circle.Circle* property), 182
- `lineRGB()` (*psychopy.visual.Form* property), 196
- `lineRGB()` (*psychopy.visual.GratingStim* property), 206
- `lineRGB()` (*psychopy.visual.ImageStim* property), 216
- `lineRGB()` (*psychopy.visual.line.Line* property), 227
- `lineRGB()` (*psychopy.visual.MovieStim* property), 237
- `lineRGB()` (*psychopy.visual.ObjMeshStim* property), 249
- `lineRGB()` (*psychopy.visual.pie.Pie* property), 260
- `lineRGB()` (*psychopy.visual.PlaneStim* property), 268
- `lineRGB()` (*psychopy.visual.polygon.Polygon* property), 276
- `lineRGB()` (*psychopy.visual.RadialStim* property), 286
- `lineRGB()` (*psychopy.visual.rect.Rect* property), 300
- `lineRGB()` (*psychopy.visual.shape.ShapeStim* property), 348
- `lineRGB()` (*psychopy.visual.SphereStim* property), 361
- `lineRGB()` (*psychopy.visual.TextBox2* property), 377
- `lineSpacing()` (*psychopy.visual.TextBox2* property), 377
- `lineWidth` (*psychopy.visual.circle.Circle* attribute), 182
- `lineWidth` (*psychopy.visual.line.Line* attribute), 227
- `lineWidth` (*psychopy.visual.pie.Pie* attribute), 260
- `lineWidth` (*psychopy.visual.polygon.Polygon* attribute), 276
- `lineWidth` (*psychopy.visual.rect.Rect* attribute), 300
- `lineWidth` (*psychopy.visual.shape.ShapeStim* attribute), 348
- `linkProgram()` (in module *psychopy.tools.gltools*), 594
- `linkProgramObjectARB()` (in module *psychopy.tools.gltools*), 595
- `listPlugins()` (in module *psychopy.plugins*), 769
- `lms()` (*psychopy.colors.Color* property), 695
- `lms2rgb()` (in module *psychopy.tools.colorspectools*), 584
- `lmsa()` (*psychopy.colors.Color* property), 695
- `load()` (*psychopy.sound.AudioClip* static method), 455
- `loadAll()` (*psychopy.preferences.Preferences* method), 772
- `loadMovie()` (*psychopy.visual.MovieStim* method), 237
- `loadMovie()` (*psychopy.visual.VlcMovieStim* method), 395
- `loadMtlFile()` (in module *psychopy.tools.gltools*), 625
- `loadObjFile()` (in module *psychopy.tools.gltools*), 623
- `loadPlugin()` (in module *psychopy.plugins*), 768

- loadUserPrefs() (*psychopy.preferences.Preferences* method), 772
- log() (*in module psychopy.logging*), 755
- log() (*psychopy.logging._Logger* method), 754
- LogFile (*class in psychopy.logging*), 754
- logged_time (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent* attribute), 550
- logged_time (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 549
- logOnFlip() (*psychopy.visual.nnlvs.VisualSystemHD* method), 411
- logOnFlip() (*psychopy.visual.rift.Rift* method), 324
- logOnFlip() (*psychopy.visual.Window* method), 429
- longName (*psychopy.hardware.crs.bits.BitsSharp* attribute), 492
- longName (*psychopy.hardware.crs.colorcal.ColorCAL* attribute), 504
- lookAt() (*in module psychopy.tools.viewtools*), 689
- loopCount() (*psychopy.visual.MovieStim* property), 237
- loopCount() (*psychopy.visual.VlcMovieStim* property), 395
- loopEnded() (*psychopy.data.ExperimentHandler* method), 698
- LS100 (*class in psychopy.hardware.minolta*), 516
- M**
- magnitude() (*psychopy.layout.Position* property), 751
- magnitude() (*psychopy.layout.Size* property), 752
- magnitude() (*psychopy.layout.Vector* property), 750
- make2DGauss() (*in module psychopy.visual.filters*), 742
- makeDKL2RGB() (*in module psychopy.monitors*), 766
- makeGauss() (*in module psychopy.visual.filters*), 742
- makeGrating() (*in module psychopy.visual.filters*), 742
- makeImageAuto() (*in module psychopy.tools.imagetools*), 634
- makeLMS2RGB() (*in module psychopy.monitors*), 767
- makeMask() (*in module psychopy.visual.filters*), 743
- makeRadialMatrix() (*in module psychopy.visual.filters*), 743
- manufacturer() (*psychopy.visual.rift.Rift* property), 324
- mapBuffer() (*in module psychopy.tools.gltools*), 615
- markerColor() (*psychopy.visual.Form* property), 197
- markerColor() (*psychopy.visual.Slider* property), 355
- markerPos (*psychopy.visual.Slider* attribute), 355
- mask (*psychopy.visual.BufferImageStim* attribute), 172
- mask (*psychopy.visual.GratingStim* attribute), 206
- mask (*psychopy.visual.ImageStim* attribute), 216
- mask (*psychopy.visual.RadialStim* attribute), 286
- maskMatrix() (*in module psychopy.visual.filters*), 743
- maskParams (*psychopy.visual.BufferImageStim* attribute), 172
- maskParams (*psychopy.visual.GratingStim* attribute), 206
- maskParams (*psychopy.visual.ImageStim* attribute), 216
- maskParams (*psychopy.visual.RadialStim* attribute), 286
- matrixFromEulerAngles() (*in module psychopy.tools.mathtools*), 660
- matrixToQuat() (*in module psychopy.tools.mathtools*), 659
- maxFlowRate() (*psychopy.hardware.qmix.Pump* property), 522
- maxRecordingSize() (*psychopy.sound.Microphone* property), 450
- mean() (*psychopy.data.QuestHandler* method), 724
- measure() (*psychopy.hardware.crs.colorcal.ColorCAL* method), 504
- measure() (*psychopy.hardware.minolta.CS100A* method), 516
- measure() (*psychopy.hardware.minolta.LS100* method), 517
- measure() (*psychopy.hardware.pr.PR650* method), 518
- measure() (*psychopy.hardware.pr.PR655* method), 520
- mergeFolder() (*in module psychopy.tools.filetools*), 587
- Method of constants, 30
- Microphone (*class in psychopy.sound*), 448
- mode() (*psychopy.data.QuestHandler* method), 724
- mode() (*psychopy.hardware.brainproducts.RemoteControlServer* property), 468
- mode() (*psychopy.hardware.crs.bits.BitsSharp* property), 492
- modelMatrix() (*psychopy.visual.RigidBodyPose* property), 338
- modifiers() (*psychopy.iohub.client.keyboard.KeyboardPress* property), 533
- modifiers() (*psychopy.iohub.client.keyboard.KeyboardRelease* property), 534
- module
- psychopy.clock, 152
 - psychopy.core, 149
 - psychopy.data, 696
 - psychopy.hardware, 464
 - psychopy.hardware.brainproducts, 467
 - psychopy.hardware.crs, 471
 - psychopy.hardware.emulator, 505
 - psychopy.hardware.forp, 510
 - psychopy.hardware.joystick, 511
 - psychopy.hardware.keyboard, 464
 - psychopy.hardware.minolta, 515

- psychopy.hardware.pr, 517
 - psychopy.hardware.qmix, 521
 - psychopy.info, 746
 - psychopy.iohub.client, 523
 - psychopy.iohub.client.keyboard, 530
 - psychopy.logging, 754
 - psychopy.misc, 759
 - psychopy.parallel, 767
 - psychopy.preferences, 772
 - psychopy.sound, 440
 - psychopy.tools, 581
 - psychopy.tools.colorspectools, 581
 - psychopy.tools.coordinatetools, 587
 - psychopy.tools.filetools, 587
 - psychopy.tools.gltools, 588
 - psychopy.tools.imagetools, 634
 - psychopy.tools.mathtools, 634
 - psychopy.tools.monitorunittools, 677
 - psychopy.tools.plottools, 678
 - psychopy.tools.typtools, 680
 - psychopy.tools.unittools, 680
 - psychopy.tools.viewtools, 682
 - psychopy.visual.filters, 741
 - psychopy.visual.windowframepack, 436
 - psychopy.visual.windowwarp, 436
 - Monitor (class in psychopy.monitors), 762
 - monitor() (psychopy.layout.Position property), 751
 - monitor() (psychopy.layout.Size property), 752
 - monitor() (psychopy.layout.Vector property), 750
 - monitorEDID() (psychopy.hardware.crs.bits.BitsSharp property), 492
 - monitorFramePeriod (psychopy.visual.Window attribute), 421
 - MonocularEyeSampleEvent (class in psychopy.iohub.devices.eyetracker), 549, 554
 - monoscopic() (psychopy.visual.nnlvs.VisualSystemHD property), 411
 - MonotonicClock (class in psychopy.clock), 153
 - MonotonicClock (class in psychopy.core), 150
 - Mouse (class in psychopy.event), 738
 - mouseMoved() (psychopy.event.Mouse method), 739
 - mouseMoveTime() (psychopy.event.Mouse method), 739
 - mouseVisible (psychopy.visual.nnlvs.VisualSystemHD attribute), 411
 - mouseVisible (psychopy.visual.rift.Rift attribute), 324
 - mouseVisible (psychopy.visual.Window attribute), 429
 - MovieStim (class in psychopy.visual), 231
 - multiFlip() (psychopy.visual.nnlvs.VisualSystemHD method), 411
 - multiFlip() (psychopy.visual.rift.Rift method), 324
 - multiplyProjectionMatrixGL() (psychopy.visual.rift.Rift method), 325
 - multiplyViewMatrixGL() (psychopy.visual.rift.Rift method), 325
 - MultiStairHandler (class in psychopy.data), 731
 - multMatrix() (in module psychopy.tools.mathtools), 663
 - multQuat() (in module psychopy.tools.mathtools), 655
 - muted() (psychopy.visual.MovieStim property), 237
- ## N
- name (psychopy.hardware.crs.bits.BitsSharp attribute), 492
 - name (psychopy.visual.Aperture attribute), 156
 - name (psychopy.visual.BufferImageStim attribute), 172
 - name (psychopy.visual.circle.Circle attribute), 182
 - name (psychopy.visual.Form attribute), 197
 - name (psychopy.visual.GratingStim attribute), 206
 - name (psychopy.visual.ImageStim attribute), 217
 - name (psychopy.visual.line.Line attribute), 227
 - name (psychopy.visual.MovieStim attribute), 237
 - name (psychopy.visual.pie.Pie attribute), 260
 - name (psychopy.visual.polygon.Polygon attribute), 276
 - name (psychopy.visual.RadialStim attribute), 286
 - name (psychopy.visual.rect.Rect attribute), 300
 - name (psychopy.visual.shape.ShapeStim attribute), 348
 - name (psychopy.visual.TextBox2 attribute), 377
 - name (psychopy.visual.TextStim attribute), 385
 - name (psychopy.visual.VlcMovieStim attribute), 395
 - name() (psychopy.hardware.emulator.ResponseEmulator property), 506
 - name() (psychopy.hardware.emulator.SyncGenerator property), 508
 - named() (psychopy.colors.Color property), 695
 - native_id() (psychopy.hardware.emulator.ResponseEmulator property), 506
 - native_id() (psychopy.hardware.emulator.SyncGenerator property), 508
 - nearClip() (psychopy.visual.nnlvs.VisualSystemHD property), 412
 - nearClip() (psychopy.visual.rift.Rift property), 325
 - nearClip() (psychopy.visual.Window property), 429
 - newCalib() (psychopy.monitors.Monitor method), 763
 - next() (psychopy.data.MultiStairHandler method), 732
 - next() (psychopy.data.PsiHandler method), 719
 - next() (psychopy.data.QuestHandler method), 724
 - next() (psychopy.data.QuestPlusHandler method), 728
 - next() (psychopy.data.StairHandler method), 715
 - next() (psychopy.data.TrialHandler method), 701
 - next() (psychopy.data.TrialHandler2 method), 705
 - next() (psychopy.data.TrialHandlerExt method), 711

- nextEditable() (psychopy.visual.nnlvs.VisualSystemHD method), 412
- nextEditable() (psychopy.visual.rift.Rift method), 325
- nextEditable() (psychopy.visual.Window method), 429
- nextEntry() (psychopy.data.ExperimentHandler method), 698
- noiseDots (psychopy.visual.DotStim attribute), 186
- NoiseStim (class in psychopy.visual), 242
- norm() (psychopy.layout.Position property), 751
- norm() (psychopy.layout.Size property), 752
- norm() (psychopy.layout.Vector property), 750
- norm() (psychopy.layout.Vertices property), 753
- normalize() (in module psychopy.tools.mathtools), 635
- normalMatrix() (in module psychopy.tools.mathtools), 666
- normalMatrix() (psychopy.visual.RigidBodyPose property), 338
- O**
- ObjMeshInfo (class in psychopy.tools.gltools), 622
- ObjMeshStim (class in psychopy.visual), 244
- OffsetVoiceKey (class in psychopy.voicekey), 775
- onResize() (psychopy.visual.nnlvs.VisualSystemHD method), 412
- onResize() (psychopy.visual.rift.Rift method), 325
- OnsetVoiceKey (class in psychopy.voicekey), 773
- opacity() (psychopy.colors.Color property), 695
- opacity() (psychopy.visual.BufferImageStim property), 172
- opacity() (psychopy.visual.circle.Circle property), 182
- opacity() (psychopy.visual.Form property), 197
- opacity() (psychopy.visual.GratingStim property), 206
- opacity() (psychopy.visual.ImageStim property), 217
- opacity() (psychopy.visual.line.Line property), 227
- opacity() (psychopy.visual.MovieStim property), 237
- opacity() (psychopy.visual.pie.Pie property), 260
- opacity() (psychopy.visual.polygon.Polygon property), 277
- opacity() (psychopy.visual.RadialStim property), 286
- opacity() (psychopy.visual.rect.Rect property), 300
- opacity() (psychopy.visual.shape.ShapeStim property), 348
- opacity() (psychopy.visual.Slider property), 355
- opacity() (psychopy.visual.TextBox2 property), 377
- opacity() (psychopy.visual.TextStim property), 385
- opacity() (psychopy.visual.VlcMovieStim property), 395
- open() (psychopy.hardware.brainproducts.RemoteControlServer method), 468
- openOutputFile() (in module psychopy.tools.filetools), 587
- openRecorder() (psychopy.hardware.brainproducts.RemoteControlServer method), 468
- ori (psychopy.visual.Aperture attribute), 157
- ori (psychopy.visual.BufferImageStim attribute), 172
- ori (psychopy.visual.circle.Circle attribute), 182
- ori (psychopy.visual.Form attribute), 197
- ori (psychopy.visual.GratingStim attribute), 206
- ori (psychopy.visual.ImageStim attribute), 217
- ori (psychopy.visual.line.Line attribute), 227
- ori (psychopy.visual.MovieStim attribute), 237
- ori (psychopy.visual.pie.Pie attribute), 260
- ori (psychopy.visual.polygon.Polygon attribute), 277
- ori (psychopy.visual.RadialStim attribute), 286
- ori (psychopy.visual.rect.Rect attribute), 300
- ori (psychopy.visual.shape.ShapeStim attribute), 348
- ori (psychopy.visual.TextBox2 attribute), 378
- ori (psychopy.visual.TextStim attribute), 386
- ori (psychopy.visual.VlcMovieStim attribute), 395
- ori() (psychopy.visual.BoxStim property), 163
- ori() (psychopy.visual.ObjMeshStim property), 249
- ori() (psychopy.visual.PlaneStim property), 268
- ori() (psychopy.visual.RigidBodyPose property), 339
- ori() (psychopy.visual.SphereStim property), 361
- ortho3Dto2D() (in module psychopy.tools.mathtools), 646
- orthogonalize() (in module psychopy.tools.mathtools), 636
- orthoProjectionMatrix() (in module psychopy.tools.viewtools), 688
- outDeviceIndex() (psychopy.sound.AudioDeviceStatus property), 463
- outputChannels() (psychopy.sound.AudioDeviceInfo property), 461
- outputLatency() (psychopy.sound.AudioDeviceInfo property), 461
- overlaps() (psychopy.visual.Aperture method), 157
- overlaps() (psychopy.visual.BufferImageStim method), 172
- overlaps() (psychopy.visual.circle.Circle method), 182
- overlaps() (psychopy.visual.Form method), 197
- overlaps() (psychopy.visual.GratingStim method), 206
- overlaps() (psychopy.visual.ImageStim method), 217
- overlaps() (psychopy.visual.line.Line method), 227
- overlaps() (psychopy.visual.MovieStim method), 237
- overlaps() (psychopy.visual.pie.Pie method), 260

- overlaps () (*psychopy.visual.polygon.Polygon method*), 277
- overlaps () (*psychopy.visual.RadialStim method*), 287
- overlaps () (*psychopy.visual.rect.Rect method*), 300
- overlaps () (*psychopy.visual.shape.ShapeStim method*), 348
- overlaps () (*psychopy.visual.TextBox2 method*), 378
- overlaps () (*psychopy.visual.TextStim method*), 386
- overlaps () (*psychopy.visual.VlcMovieStim method*), 395
- overlayPresent () (*psychopy.visual.rift.Rift property*), 325
- overwriteProtection () (*psychopy.hardware.brainproducts.RemoteControlServer property*), 468
- ## P
- padding () (*psychopy.visual.TextBox2 property*), 378
- palette () (*psychopy.visual.TextBox2 property*), 378
- pallette () (*psychopy.visual.TextBox2 property*), 378
- ParallelPort (*in module psychopy.parallel*), 767
- paramEstimate () (*psychopy.data.QuestPlusHandler property*), 729
- parseSpectrumOutput () (*psychopy.hardware.pr.PR650 method*), 518
- parseSpectrumOutput () (*psychopy.hardware.pr.PR655 method*), 520
- participant () (*psychopy.hardware.brainproducts.RemoteControlServer property*), 468
- PatchStim (*class in psychopy.visual*), 251
- pause () (*psychopy.hardware.crs.bits.BitsSharp method*), 492
- pause () (*psychopy.sound.backend_ptb.SoundPTB method*), 443
- pause () (*psychopy.sound.backend_sounddevice.SoundDeviceSound method*), 445
- pause () (*psychopy.sound.Microphone method*), 451
- pause () (*psychopy.visual.MovieStim method*), 238
- pause () (*psychopy.visual.VlcMovieStim method*), 396
- pauseRecording () (*psychopy.hardware.brainproducts.RemoteControlServer method*), 469
- peak_velocity_xy (*psychopy.iohub.devices.eyetracker.FixationEndEvent attribute*), 559
- peak_velocity_xy (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent attribute*), 561
- percentageComplete () (*psychopy.visual.VlcMovieStim property*), 396
- perfHudMode () (*psychopy.visual.rift.Rift method*), 325
- perp () (*in module psychopy.tools.mathtools*), 639
- perspectiveProjectionMatrix () (*in module psychopy.tools.viewtools*), 688
- phase (*psychopy.visual.GratingStim attribute*), 207
- phase (*psychopy.visual.RadialStim attribute*), 287
- Pie (*class in psychopy.visual.pie*), 255
- pix () (*psychopy.layout.Position property*), 751
- pix () (*psychopy.layout.Size property*), 752
- pix () (*psychopy.layout.Vector property*), 750
- pix () (*psychopy.layout.Vertices property*), 753
- pix2cm () (*in module psychopy.tools.monitorunittools*), 678
- pix2deg () (*in module psychopy.tools.monitorunittools*), 678
- pixelsPerTanAngleAtCenter () (*psychopy.visual.rift.Rift property*), 326
- PlaneStim (*class in psychopy.visual*), 264
- play () (*psychopy.sound.backend_ptb.SoundPTB method*), 443
- play () (*psychopy.sound.backend_pygame.SoundPygame method*), 448
- play () (*psychopy.sound.backend_pyo.SoundPyo method*), 446
- play () (*psychopy.sound.backend_sounddevice.SoundDeviceSound method*), 445
- play () (*psychopy.visual.MovieStim method*), 238
- play () (*psychopy.visual.VlcMovieStim method*), 396
- playback () (*psychopy.microphone.AdvAudioCapture method*), 758
- playMarker () (*psychopy.microphone.AdvAudioCapture method*), 758
- plotFrameIntervals () (*in module psychopy.tools.plottools*), 678
- pointInPolygon () (*in module psychopy.visual.helpers*), 210
- pointToNdc () (*in module psychopy.tools.viewtools*), 689
- pol2cart () (*in module psychopy.tools.coordinatetools*), 587
- poll () (*psychopy.sound.Microphone method*), 451
- pollStatus () (*psychopy.hardware.crs.bits.BitsSharp method*), 492
- Polygon (*class in psychopy.visual.polygon*), 271
- polygonsOverlap () (*in module psychopy.visual.helpers*), 210
- pos () (*psychopy.layout.Vertices property*), 753
- pos () (*psychopy.visual.Aperture property*), 157
- pos () (*psychopy.visual.BoxStim property*), 163
- pos () (*psychopy.visual.BufferImageStim property*), 172
- pos () (*psychopy.visual.circle.Circle property*), 182
- pos () (*psychopy.visual.Form property*), 197
- pos () (*psychopy.visual.GratingStim property*), 207
- pos () (*psychopy.visual.ImageStim property*), 217
- pos () (*psychopy.visual.LightSource property*), 221

- pos () (*psychopy.visual.line.Line* property), 228
- pos () (*psychopy.visual.MovieStim* property), 238
- pos () (*psychopy.visual.ObjMeshStim* property), 249
- pos () (*psychopy.visual.pie.Pie* property), 261
- pos () (*psychopy.visual.PlaneStim* property), 268
- pos () (*psychopy.visual.polygon.Polygon* property), 277
- pos () (*psychopy.visual.RadialStim* property), 287
- pos () (*psychopy.visual.rect.Rect* property), 300
- pos () (*psychopy.visual.RigidBodyPose* property), 339
- pos () (*psychopy.visual.shape.ShapeStim* property), 348
- pos () (*psychopy.visual.Slider* property), 355
- pos () (*psychopy.visual.SphereStim* property), 361
- pos () (*psychopy.visual.TextBox2* property), 378
- pos () (*psychopy.visual.TextStim* property), 386
- pos () (*psychopy.visual.VlcMovieStim* property), 396
- Position (class in *psychopy.layout*), 750
- positionSecs () (*psychopy.sound.AudioDeviceStatus* property), 463
- posOri () (*psychopy.visual.RigidBodyPose* property), 339
- posOriToMatrix () (in module *psychopy.tools.mathtools*), 669
- posPix () (*psychopy.visual.Aperture* property), 157
- posPix () (*psychopy.visual.TextStim* property), 386
- posterior () (*psychopy.data.QuestPlusHandler* property), 729
- ppd_x (*psychopy.iohub.devices.eyetracker.FixationStartEvent* attribute), 557
- ppd_x (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 555
- ppd_x (*psychopy.iohub.devices.eyetracker.SaccadeStartEvent* attribute), 559
- ppd_y (*psychopy.iohub.devices.eyetracker.FixationStartEvent* attribute), 557
- ppd_y (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 555
- ppd_y (*psychopy.iohub.devices.eyetracker.SaccadeStartEvent* attribute), 559
- PR650 (class in *psychopy.hardware.pr*), 517
- PR655 (class in *psychopy.hardware.pr*), 519
- predictedLatency () (*psychopy.sound.AudioDeviceStatus* property), 463
- Preferences (class in *psychopy.preferences*), 772
- pressEventID () (*psychopy.iohub.client.keyboard.KeyboardRelease* property), 534
- primeClock () (*psychopy.hardware.crs.bits.BitsPlusPlus* method), 473
- primeClock () (*psychopy.hardware.crs.bits.BitsSharp* method), 492
- printAsText () (*psychopy.data.MultiStairHandler* method), 732
- printAsText () (*psychopy.data.PsiHandler* method), 719
- printAsText () (*psychopy.data.QuestHandler* method), 724
- printAsText () (*psychopy.data.QuestPlusHandler* method), 729
- printAsText () (*psychopy.data.StairHandler* method), 715
- printAsText () (*psychopy.data.TrialHandler* method), 701
- printAsText () (*psychopy.data.TrialHandler2* method), 705
- printAsText () (*psychopy.data.TrialHandlerExt* method), 711
- prior () (*psychopy.data.QuestPlusHandler* property), 729
- productName () (*psychopy.visual.rift.Rift* property), 326
- project () (in module *psychopy.tools.mathtools*), 639
- projectFrustum () (in module *psychopy.tools.viewtools*), 685
- projectFrustumToPlane () (in module *psychopy.tools.viewtools*), 686
- projectionMatrix () (*psychopy.visual.nnlvs.VisualSystemHD* property), 412
- projectionMatrix () (*psychopy.visual.rift.Rift* property), 326
- projectionMatrix () (*psychopy.visual.Window* property), 429
- projectorFramePacker (class in *psychopy.visual.windowframepack*), 436
- PsiHandler (class in *psychopy.data*), 717
- psychopy.clock
- psychopy.core
- psychopy.data
- psychopy.hardware
- psychopy.hardware.brainproducts
- psychopy.hardware.crs
- psychopy.hardware.emulator
- psychopy.hardware.forp
- psychopy.hardware.joystick
- psychopy.hardware.keyboard

psychopy.hardware.minolta
 module, 515
 psychopy.hardware.pr
 module, 517
 psychopy.hardware.qmix
 module, 521
 psychopy.info
 module, 746
 psychopy.iohub.client
 module, 523
 psychopy.iohub.client.keyboard
 module, 530
 psychopy.logging
 module, 754
 psychopy.misc
 module, 759
 psychopy.parallel
 module, 767
 psychopy.preferences
 module, 772
 psychopy.sound
 module, 440
 psychopy.tools
 module, 581
 psychopy.tools.colorspectools
 module, 581
 psychopy.tools.coordinatetools
 module, 587
 psychopy.tools.filetools
 module, 587
 psychopy.tools.gltools
 module, 588
 psychopy.tools.imagetools
 module, 634
 psychopy.tools.mathtools
 module, 634
 psychopy.tools.monitorunittools
 module, 677
 psychopy.tools.plottools
 module, 678
 psychopy.tools.typtools
 module, 680
 psychopy.tools.unittools
 module, 680
 psychopy.tools.viewtools
 module, 682
 psychopy.visual.filters
 module, 741
 psychopy.visual.windowframepack
 module, 436
 psychopy.visual.windowwarp
 module, 436
 pt () (*psychopy.layout.Position property*), 751
 pt () (*psychopy.layout.Size property*), 752
 pt () (*psychopy.layout.Vector property*), 750
 pts () (*psychopy.visual.MovieStim property*), 238
 Pump (*class in psychopy.hardware.qmix*), 521
 pupil_measure1 (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent attribute*), 550
 pupil_measure1_type (*psychopy.iohub.devices.eyetracker.FixationStartEvent attribute*), 557
 pupil_measure1_type (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent attribute*), 550, 555
 pupil_measure1_type (*psychopy.iohub.devices.eyetracker.SaccadeStartEvent attribute*), 559
 pupil_measure2 (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent attribute*), 550
 pupil_measure2_type (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute*), 551
 pupil_measure2_type (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent attribute*), 550
 pupil_measure_1 (*psychopy.iohub.devices.eyetracker.FixationStartEvent attribute*), 557
 pupil_measure_1 (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent attribute*), 554
 pupil_measure_1 (*psychopy.iohub.devices.eyetracker.SaccadeStartEvent attribute*), 559

Q
 quantile () (*psychopy.data.QuestHandler method*), 724
 quatFromAxisAngle () (*in module psychopy.tools.mathtools*), 652
 quatMagnitude () (*in module psychopy.tools.mathtools*), 654
 quatToAxisAngle () (*in module psychopy.tools.mathtools*), 651
 quatToMatrix () (*in module psychopy.tools.mathtools*), 658
 quatYawPitchRoll () (*in module psychopy.tools.mathtools*), 653
 QueryObjectInfo (*class in psychopy.tools.gltools*), 598
 QuestHandler (*class in psychopy.data*), 721
 QuestPlusHandler (*class in psychopy.data*), 726
 quit () (*psychopy.iohub.client.ioHubConnection method*), 529
 quitApp () (*in module psychopy.app*), 693

R

- radialCycles (*psychopy.visual.RadialStim* attribute), 287
- radialPhase (*psychopy.visual.RadialStim* attribute), 287
- RadialStim (class in *psychopy.visual*), 281
- radians () (in module *psychopy.tools.unittools*), 680
- radius (*psychopy.visual.circle.Circle* attribute), 178, 183
- radius (*psychopy.visual.pie.Pie* attribute), 256, 261
- radius (*psychopy.visual.polygon.Polygon* attribute), 277
- range () (*psychopy.data.QuestHandler* property), 724
- rating () (*psychopy.visual.Slider* property), 355
- RatingScale (class in *psychopy.visual*), 291
- raw_x (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 550, 554
- raw_y (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 550, 554
- read () (*psychopy.hardware.crs.bits.BitsSharp* method), 493
- readline () (*psychopy.hardware.crs.colorcal.ColorCAL* method), 505
- readPin () (*psychopy.parallel* method), 768
- readSecs () (*psychopy.sound.AudioDeviceStatus* property), 463
- rec709TF () (in module *psychopy.tools.colors spacetools*), 585
- recBufferSecs () (*psychopy.sound.Microphone* property), 451
- recenterTrackingOrigin () (*psychopy.visual.rift.Rift* method), 326
- record () (*psychopy.microphone.AdvAudioCapture* method), 758
- record () (*psychopy.sound.Microphone* method), 451
- recordedSecs () (*psychopy.sound.AudioDeviceStatus* property), 463
- recordFrameIntervals (*psychopy.visual.nnlvs.VisualSystemHD* attribute), 412
- recordFrameIntervals (*psychopy.visual.rift.Rift* attribute), 326
- recordFrameIntervals (*psychopy.visual.Window* attribute), 429
- recording () (*psychopy.sound.Microphone* property), 451
- recordRating () (*psychopy.visual.Slider* method), 355
- Rect (class in *psychopy.visual.rect*), 295
- reflect () (in module *psychopy.tools.mathtools*), 637
- RemoteControlServer (class in *psychopy.hardware.brainproducts*), 467
- removeEditable () (*psychopy.visual.nnlvs.VisualSystemHD* method), 413
- removeEditable () (*psychopy.visual.rift.Rift* method), 326
- removeEditable () (*psychopy.visual.Window* method), 430
- removeTarget () (*psychopy.logging._Logger* method), 755
- render () (*psychopy.colors.Color* method), 695
- replay () (*psychopy.visual.MovieStim* method), 238
- replay () (*psychopy.visual.VlcMovieStim* method), 396
- reporting () (*psychopy.iohub.client.keyboard.Keyboard* property), 531
- requestedStartTime () (*psychopy.sound.AudioDeviceStatus* property), 463
- requestedStopTime () (*psychopy.sound.AudioDeviceStatus* property), 463
- requireInternetAccess () (in module *psychopy.web*), 776
- resample () (*psychopy.microphone.AdvAudioCapture* method), 758
- rescaleColor () (in module *psychopy.tools.colors spacetools*), 586
- reset () (*psychopy.clock.Clock* method), 152
- reset () (*psychopy.clock.CountdownTimer* method), 153
- reset () (*psychopy.core.Clock* method), 149
- reset () (*psychopy.core.CountdownTimer* method), 150
- reset () (*psychopy.hardware.crs.bits.BitsPlusPlus* method), 473
- reset () (*psychopy.hardware.crs.bits.BitsSharp* method), 493
- reset () (*psychopy.microphone.AdvAudioCapture* method), 758
- reset () (*psychopy.visual.Form* method), 197
- reset () (*psychopy.visual.Slider* method), 355
- reset () (*psychopy.visual.TextBox2* method), 378
- resetClock () (*psychopy.hardware.crs.bits.BitsPlusPlus* method), 473
- resetClock () (*psychopy.hardware.crs.bits.BitsSharp* method), 493
- resetEyeTransform () (*psychopy.visual.nnlvs.VisualSystemHD* method), 413
- resetEyeTransform () (*psychopy.visual.rift.Rift* method), 326
- resetEyeTransform () (*psychopy.visual.Window* method), 430
- resetPrefs () (*psychopy.preferences.Preferences* method), 772

resetViewport () (psychopy.visual.nnlvs.VisualSystemHD method), 413
 resetViewport () (psychopy.visual.rift.Rift method), 327
 resetViewport () (psychopy.visual.Window method), 430
 responseColor () (psychopy.visual.Form property), 197
 ResponseEmulator (class in psychopy.hardware.emulator), 505, 509
 restoreBadPrefs () (psychopy.preferences.Preferences method), 772
 resumeRecording () (psychopy.hardware.brainproducts.RemoteControlServer method), 469
 reverseProject () (in module psychopy.tools.mathtools), 667
 rewind () (psychopy.visual.MovieStim method), 239
 rewind () (psychopy.visual.VlcMovieStim method), 397
 rgb () (psychopy.colors.Color property), 695
 RGB () (psychopy.visual.BoxStim property), 160
 RGB () (psychopy.visual.BufferImageStim property), 167
 RGB () (psychopy.visual.circle.Circle property), 178
 RGB () (psychopy.visual.Form property), 191
 RGB () (psychopy.visual.GratingStim property), 201
 RGB () (psychopy.visual.ImageStim property), 212
 RGB () (psychopy.visual.line.Line property), 223
 RGB () (psychopy.visual.MovieStim property), 232
 rgb () (psychopy.visual.nnlvs.VisualSystemHD property), 413
 RGB () (psychopy.visual.ObjMeshStim property), 245
 RGB () (psychopy.visual.pie.Pie property), 256
 RGB () (psychopy.visual.PlaneStim property), 264
 RGB () (psychopy.visual.polygon.Polygon property), 273
 RGB () (psychopy.visual.RadialStim property), 281
 RGB () (psychopy.visual.rect.Rect property), 296
 rgb () (psychopy.visual.rift.Rift property), 327
 RGB () (psychopy.visual.shape.ShapeStim property), 344
 RGB () (psychopy.visual.SphereStim property), 357
 RGB () (psychopy.visual.TextBox2 property), 373
 RGB () (psychopy.visual.TextStim property), 381
 rgb () (psychopy.visual.Window property), 430
 rgb1 () (psychopy.colors.Color property), 695
 rgb255 () (psychopy.colors.Color property), 695
 rgb2dtk1Cart () (in module psychopy.tools.colorspectools), 583
 rgb2hsv () (in module psychopy.tools.colorspectools), 584
 rgb2lms () (in module psychopy.tools.colorspectools), 585
 rgba () (psychopy.colors.Color property), 695
 rgba1 () (psychopy.colors.Color property), 695
 rgba255 () (psychopy.colors.Color property), 695
 Rift (class in psychopy.visual.rift), 305
 right_angle_x (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 551, 556
 right_angle_y (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 552, 556
 right_eye_cam_x (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 551, 571
 right_eye_cam_y (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 551, 571
 right_eye_cam_z (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 551, 571
 right_gaze_x (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 539, 551, 556, 571
 right_gaze_y (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 539, 551, 556, 571
 right_gaze_z (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 551
 right_ppd_x (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 556
 right_ppd_y (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 556
 right_pupil_measure1 (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 552
 right_pupil_measure1_type (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 552, 556
 right_pupil_measure2 (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 552
 right_pupil_measure2_type (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 552
 right_pupil_measure_1 (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 540, 556, 571
 right_raw_x (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 539, 552, 556
 right_raw_y (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 539, 552, 556
 right_velocity_x (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 556
 right_velocity_xy (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 556

- attribute*), 556
- `right_velocity_y` (*psy-*
chopy.iohub.devices.eyetracker.BinocularEyeSampleEvent
attribute), 556
- `RigidBodyPose` (*class in psychopy.visual*), 336
- `rms()` (*in module psychopy.voicekey*), 775
- `rms()` (*psychopy.sound.AudioClip method*), 455
- `rotationMatrix()` (*in module psy-*
chopy.tools.mathtools), 661
- `RTBoxAddKeys()` (*psy-*
chopy.hardware.crs.bits.BitsSharp method),
479
- `RTBoxCalibrate()` (*psy-*
chopy.hardware.crs.bits.BitsSharp method),
479
- `RTBoxClear()` (*psychopy.hardware.crs.bits.BitsSharp*
method), 480
- `RTBoxDisable()` (*psy-*
chopy.hardware.crs.bits.BitsSharp method),
480
- `RTBoxEnable()` (*psy-*
chopy.hardware.crs.bits.BitsSharp method),
480
- `RTBoxKeysPressed()` (*psy-*
chopy.hardware.crs.bits.BitsSharp method),
481
- `RTBoxResetKeys()` (*psy-*
chopy.hardware.crs.bits.BitsSharp method),
481
- `RTBoxSetKeys()` (*psy-*
chopy.hardware.crs.bits.BitsSharp method),
481
- `RTBoxWait()` (*psychopy.hardware.crs.bits.BitsSharp*
method), 482
- `RTBoxWaitN()` (*psychopy.hardware.crs.bits.BitsSharp*
method), 482
- `run()` (*psychopy.hardware.emulator.ResponseEmulator*
method), 506, 509
- `run()` (*psychopy.hardware.emulator.SyncGenerator*
method), 508, 510
- `runSetupProcedure()` (*psy-*
chopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker
method), 538
- `runSetupProcedure()` (*psy-*
chopy.iohub.devices.eyetracker.hw.mouse.EyeTracker
method), 577
- `runSetupProcedure()` (*psy-*
chopy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core.EyeTracker
method), 548
- `runSetupProcedure()` (*psy-*
chopy.iohub.devices.eyetracker.hw.tobii.EyeTracker
method), 570
- `RunTimeInfo` (*class in psychopy.info*), 746
- ## S
- `SaccadeEndEvent` (*class in psy-*
chopy.iohub.devices.eyetracker), 560
- `SaccadeStartEvent` (*class in psy-*
chopy.iohub.devices.eyetracker), 559
- `sampleRate()` (*psychopy.sound.AudioDeviceStatus*
property), 463
- `sampleRateHz()` (*psychopy.sound.AudioClip prop-*
erty), 455
- `samples()` (*psychopy.sound.AudioClip property*), 456
- `samples_from_file()` (*in module psy-*
chopy.voicekey), 776
- `samples_from_table()` (*in module psy-*
chopy.voicekey), 776
- `samples_to_file()` (*in module psychopy.voicekey*),
776
- `save()` (*psychopy.monitors.Monitor method*), 763
- `save()` (*psychopy.sound.AudioClip method*), 456
- `save()` (*psychopy.voicekey.OnsetVoiceKey method*),
774
- `saveAppData()` (*psychopy.preferences.Preferences*
method), 772
- `saveAsExcel()` (*psychopy.data.MultiStairHandler*
method), 733
- `saveAsExcel()` (*psychopy.data.PsiHandler method*),
719
- `saveAsExcel()` (*psychopy.data.QuestHandler*
method), 724
- `saveAsExcel()` (*psychopy.data.QuestPlusHandler*
method), 729
- `saveAsExcel()` (*psychopy.data.StairHandler*
method), 715
- `saveAsExcel()` (*psychopy.data.TrialHandler*
method), 701
- `saveAsExcel()` (*psychopy.data.TrialHandler2*
method), 705
- `saveAsExcel()` (*psychopy.data.TrialHandlerExt*
method), 711
- `saveAsJson()` (*psychopy.data.MultiStairHandler*
method), 733
- `saveAsJson()` (*psychopy.data.PsiHandler method*),
719
- `saveAsJson()` (*psychopy.data.QuestHandler*
method), 725
- `saveAsJson()` (*psychopy.data.QuestPlusHandler*
method), 730
- `saveAsJson()` (*psychopy.data.StairHandler method*),
719
- `saveAsJson()` (*psychopy.data.TrialHandler method*),
702
- `saveAsJson()` (*psychopy.data.TrialHandler2*
method), 706
- `saveAsJson()` (*psychopy.data.TrialHandlerExt*
method), 712

- saveAsPickle()* (*psychopy.data.ExperimentHandler method*), 698
saveAsPickle() (*psychopy.data.MultiStairHandler method*), 734
saveAsPickle() (*psychopy.data.PsiHandler method*), 720
saveAsPickle() (*psychopy.data.QuestHandler method*), 725
saveAsPickle() (*psychopy.data.QuestPlusHandler method*), 730
saveAsPickle() (*psychopy.data.StairHandler method*), 716
saveAsPickle() (*psychopy.data.TrialHandler method*), 702
saveAsPickle() (*psychopy.data.TrialHandler2 method*), 707
saveAsPickle() (*psychopy.data.TrialHandlerExt method*), 712
saveAsText() (*psychopy.data.MultiStairHandler method*), 734
saveAsText() (*psychopy.data.PsiHandler method*), 720
saveAsText() (*psychopy.data.QuestHandler method*), 725
saveAsText() (*psychopy.data.QuestPlusHandler method*), 730
saveAsText() (*psychopy.data.StairHandler method*), 716
saveAsText() (*psychopy.data.TrialHandler method*), 702
saveAsText() (*psychopy.data.TrialHandler2 method*), 707
saveAsText() (*psychopy.data.TrialHandlerExt method*), 712
saveAsWideText() (*psychopy.data.ExperimentHandler method*), 698
saveAsWideText() (*psychopy.data.TrialHandler method*), 703
saveAsWideText() (*psychopy.data.TrialHandler2 method*), 707
saveAsWideText() (*psychopy.data.TrialHandlerExt method*), 712
saveFrameIntervals() (*psychopy.visual.nnivs.VisualSystemHD method*), 413
saveFrameIntervals() (*psychopy.visual.rift.Rift method*), 327
saveFrameIntervals() (*psychopy.visual.Window method*), 430
saveMon() (*psychopy.monitors.Monitor method*), 763
saveMovieFrames() (*psychopy.visual.nnivs.VisualSystemHD method*), 413
saveMovieFrames() (*psychopy.visual.rift.Rift method*), 327
saveMovieFrames() (*psychopy.visual.Window method*), 431
savePosterior() (*psychopy.data.PsiHandler method*), 720
saveUserPrefs() (*psychopy.preferences.Preferences method*), 772
sawtooth() (*psychopy.sound.AudioClip static method*), 456
scale() (*in module psychopy.tools.mathtools*), 648
scaleMatrix() (*in module psychopy.tools.mathtools*), 661
SceneSkybox (*class in psychopy.visual*), 340
schedulePosition() (*psychopy.sound.AudioDeviceStatus property*), 464
scissor() (*psychopy.visual.nnivs.VisualSystemHD property*), 414
scissor() (*psychopy.visual.rift.Rift property*), 328
scissor() (*psychopy.visual.Window property*), 431
scissorTest() (*psychopy.visual.nnivs.VisualSystemHD property*), 414
scissorTest() (*psychopy.visual.rift.Rift property*), 328
scissorTest() (*psychopy.visual.Window property*), 431
screenshot() (*psychopy.visual.nnivs.VisualSystemHD property*), 414
screenshot() (*psychopy.visual.rift.Rift property*), 328
screenshot() (*psychopy.visual.Window property*), 431
scrollbarWidth() (*psychopy.visual.Form property*), 198
sd() (*psychopy.data.QuestHandler method*), 725
seek() (*psychopy.visual.MovieStim method*), 239
seek() (*psychopy.visual.VlcMovieStim method*), 397
sendAnalog() (*psychopy.hardware.crs.bits.BitsSharp method*), 493
sendAnnotation() (*psychopy.hardware.brainproducts.RemoteControlServer method*), 469
sendMessage() (*psychopy.hardware.crs.bits.BitsSharp method*), 494
sendMessage() (*psychopy.hardware.crs.colorcal.ColorCAL method*), 505
sendMessage() (*psychopy.hardware.minolta.CS100A method*), 516
sendMessage() (*psychopy.hardware.minolta.LS100*

method), 517
 sendMessage () (*psychopy.hardware.pr.PR650 method*), 518
 sendMessageEvent () (*psychopy.iohub.client.ioHubConnection method*), 526
 sendRaw () (*psychopy.hardware.brainproducts.RemoteControlServer method*), 469
 sendTrigger () (*psychopy.hardware.crs.bits.BitsPlusPlus method*), 473
 sendTrigger () (*psychopy.hardware.crs.bits.BitsSharp method*), 494
 sensorSampleTime () (*psychopy.visual.rift.Rift property*), 328
 serialNumber () (*psychopy.visual.rift.Rift property*), 328
 set () (*psychopy.colors.Color method*), 695
 set () (*psychopy.layout.Position method*), 751
 set () (*psychopy.layout.Size method*), 752
 set () (*psychopy.layout.Vector method*), 750
 setAmbientLight () (*in module psychopy.tools.gltools*), 621
 setAnalog () (*psychopy.hardware.crs.bits.BitsSharp method*), 494
 setAnchor () (*psychopy.visual.Aperture method*), 157
 setAnchor () (*psychopy.visual.BoxStim method*), 163
 setAnchor () (*psychopy.visual.BufferImageStim method*), 173
 setAnchor () (*psychopy.visual.Form method*), 198
 setAnchor () (*psychopy.visual.GratingStim method*), 207
 setAnchor () (*psychopy.visual.ImageStim method*), 217
 setAnchor () (*psychopy.visual.MovieStim method*), 239
 setAnchor () (*psychopy.visual.ObjMeshStim method*), 249
 setAnchor () (*psychopy.visual.PlaneStim method*), 268
 setAnchor () (*psychopy.visual.RadialStim method*), 287
 setAnchor () (*psychopy.visual.SphereStim method*), 361
 setAnchor () (*psychopy.visual.TextBox2 method*), 378
 setAnchor () (*psychopy.visual.VlcMovieStim method*), 397
 setAngularCycles () (*psychopy.visual.RadialStim method*), 287
 setAngularPhase () (*psychopy.visual.RadialStim method*), 288
 setas () (*psychopy.layout.Vertices method*), 753
 setAutoDraw () (*psychopy.visual.Aperture method*), 157
 setAutoDraw () (*psychopy.visual.BufferImageStim method*), 173
 setAutoDraw () (*psychopy.visual.circle.Circle method*), 183
 setAutoDraw () (*psychopy.visual.Form method*), 198
 setAutoDraw () (*psychopy.visual.GratingStim method*), 207
 setAutoDraw () (*psychopy.visual.ImageStim method*), 217
 setAutoDraw () (*psychopy.visual.line.Line method*), 228
 setAutoDraw () (*psychopy.visual.MovieStim method*), 239
 setAutoDraw () (*psychopy.visual.pie.Pie method*), 261
 setAutoDraw () (*psychopy.visual.polygon.Polygon method*), 277
 setAutoDraw () (*psychopy.visual.RadialStim method*), 288
 setAutoDraw () (*psychopy.visual.rect.Rect method*), 301
 setAutoDraw () (*psychopy.visual.shape.ShapeStim method*), 349
 setAutoDraw () (*psychopy.visual.TextBox2 method*), 378
 setAutoDraw () (*psychopy.visual.TextStim method*), 386
 setAutoDraw () (*psychopy.visual.VlcMovieStim method*), 397
 setAutoLog () (*psychopy.visual.Aperture method*), 157
 setAutoLog () (*psychopy.visual.BufferImageStim method*), 173
 setAutoLog () (*psychopy.visual.circle.Circle method*), 183
 setAutoLog () (*psychopy.visual.Form method*), 198
 setAutoLog () (*psychopy.visual.GratingStim method*), 207
 setAutoLog () (*psychopy.visual.ImageStim method*), 217
 setAutoLog () (*psychopy.visual.line.Line method*), 228
 setAutoLog () (*psychopy.visual.MovieStim method*), 239
 setAutoLog () (*psychopy.visual.pie.Pie method*), 261
 setAutoLog () (*psychopy.visual.polygon.Polygon method*), 277
 setAutoLog () (*psychopy.visual.RadialStim method*), 288
 setAutoLog () (*psychopy.visual.rect.Rect method*), 301
 setAutoLog () (*psychopy.visual.shape.ShapeStim method*), 349

setAutoLog () (*psychopy.visual.TextBox* method), 369
 setAutoLog () (*psychopy.visual.TextBox2* method), 378
 setAutoLog () (*psychopy.visual.TextStim* method), 386
 setAutoLog () (*psychopy.visual.VlcMovieStim* method), 397
 setBackColor () (*psychopy.visual.BoxStim* method), 163
 setBackColor () (*psychopy.visual.BufferImageStim* method), 173
 setBackColor () (*psychopy.visual.Form* method), 198
 setBackColor () (*psychopy.visual.GratingStim* method), 207
 setBackColor () (*psychopy.visual.ImageStim* method), 218
 setBackColor () (*psychopy.visual.MovieStim* method), 239
 setBackColor () (*psychopy.visual.ObjMeshStim* method), 249
 setBackColor () (*psychopy.visual.PlaneStim* method), 268
 setBackColor () (*psychopy.visual.RadialStim* method), 288
 setBackColor () (*psychopy.visual.SphereStim* method), 361
 setBackColor () (*psychopy.visual.TextBox2* method), 378
 setBackend () (*psychopy.hardware.keyboard.Keyboard* class method), 465
 setBackgroundColor () (*psychopy.visual.TextBox* method), 370
 setBackRGB () (*psychopy.visual.BoxStim* method), 163
 setBackRGB () (*psychopy.visual.BufferImageStim* method), 173
 setBackRGB () (*psychopy.visual.circle.Circle* method), 183
 setBackRGB () (*psychopy.visual.Form* method), 198
 setBackRGB () (*psychopy.visual.GratingStim* method), 207
 setBackRGB () (*psychopy.visual.ImageStim* method), 218
 setBackRGB () (*psychopy.visual.line.Line* method), 228
 setBackRGB () (*psychopy.visual.MovieStim* method), 239
 setBackRGB () (*psychopy.visual.ObjMeshStim* method), 249
 setBackRGB () (*psychopy.visual.pie.Pie* method), 261
 setBackRGB () (*psychopy.visual.PlaneStim* method), 268
 setBackRGB () (*psychopy.visual.polygon.Polygon* method), 278
 setBackRGB () (*psychopy.visual.RadialStim* method), 288
 setBackRGB () (*psychopy.visual.rect.Rect* method), 301
 setBackRGB () (*psychopy.visual.shape.ShapeStim* method), 349
 setBackRGB () (*psychopy.visual.SphereStim* method), 361
 setBackRGB () (*psychopy.visual.TextBox2* method), 379
 setBlendmode () (*psychopy.visual.GratingStim* method), 207
 setBlendMode () (*psychopy.visual.nnivs.VisualSystemHD* method), 414
 setBlendmode () (*psychopy.visual.RadialStim* method), 288
 setBlendMode () (*psychopy.visual.rift.Rift* method), 328
 setBorderColor () (*psychopy.visual.BoxStim* method), 163
 setBorderColor () (*psychopy.visual.BufferImageStim* method), 173
 setBorderColor () (*psychopy.visual.circle.Circle* method), 183
 setBorderColor () (*psychopy.visual.Form* method), 198
 setBorderColor () (*psychopy.visual.GratingStim* method), 207
 setBorderColor () (*psychopy.visual.ImageStim* method), 218
 setBorderColor () (*psychopy.visual.line.Line* method), 228
 setBorderColor () (*psychopy.visual.MovieStim* method), 239
 setBorderColor () (*psychopy.visual.ObjMeshStim* method), 249
 setBorderColor () (*psychopy.visual.pie.Pie* method), 261
 setBorderColor () (*psychopy.visual.PlaneStim* method), 268
 setBorderColor () (*psychopy.visual.polygon.Polygon* method), 278
 setBorderColor () (*psychopy.visual.RadialStim* method), 288
 setBorderColor () (*psychopy.visual.rect.Rect* method), 301
 setBorderColor () (*psychopy.visual.shape.ShapeStim* method), 349
 setBorderColor () (*psychopy.visual.SphereStim* method), 361
 setBorderColor () (*psychopy.visual.TextBox*

- method*), 370
- setBorderColor() (*psychopy.visual.TextBox2 method*), 379
- setBorderColor() (*psychopy.visual.BoxStim method*), 164
- setBorderColor() (*psychopy.visual.BufferImageStim method*), 173
- setBorderColor() (*psychopy.visual.circle.Circle method*), 183
- setBorderColor() (*psychopy.visual.Form method*), 198
- setBorderColor() (*psychopy.visual.GratingStim method*), 207
- setBorderColor() (*psychopy.visual.ImageStim method*), 218
- setBorderColor() (*psychopy.visual.line.Line method*), 228
- setBorderColor() (*psychopy.visual.MovieStim method*), 239
- setBorderColor() (*psychopy.visual.ObjMeshStim method*), 249
- setBorderColor() (*psychopy.visual.pie.Pie method*), 261
- setBorderColor() (*psychopy.visual.PlaneStim method*), 269
- setBorderColor() (*psychopy.visual.polygon.Polygon method*), 278
- setBorderColor() (*psychopy.visual.RadialStim method*), 288
- setBorderColor() (*psychopy.visual.rect.Rect method*), 301
- setBorderColor() (*psychopy.visual.shape.ShapeStim method*), 349
- setBorderColor() (*psychopy.visual.SphereStim method*), 361
- setBorderColor() (*psychopy.visual.TextBox2 method*), 379
- setBorderWidth() (*psychopy.visual.TextBox method*), 370
- setBuffer() (*psychopy.visual.nnivs.VisualSystemHD method*), 414
- setBuffer() (*psychopy.visual.rift.Rift method*), 328
- setBuffer() (*psychopy.visual.Window method*), 432
- setCalibDate() (*psychopy.monitors.Monitor method*), 763
- setColor() (*psychopy.visual.BoxStim method*), 164
- setColor() (*psychopy.visual.BufferImageStim method*), 173
- setColor() (*psychopy.visual.circle.Circle method*), 183
- setColor() (*psychopy.visual.Form method*), 198
- setColor() (*psychopy.visual.GratingStim method*), 207
- setColor() (*psychopy.visual.ImageStim method*), 218
- setColor() (*psychopy.visual.line.Line method*), 228
- setColor() (*psychopy.visual.MovieStim method*), 239
- setColor() (*psychopy.visual.nnivs.VisualSystemHD method*), 415
- setColor() (*psychopy.visual.ObjMeshStim method*), 249
- setColor() (*psychopy.visual.pie.Pie method*), 261
- setColor() (*psychopy.visual.PlaneStim method*), 269
- setColor() (*psychopy.visual.polygon.Polygon method*), 278
- setColor() (*psychopy.visual.RadialStim method*), 288
- setColor() (*psychopy.visual.rect.Rect method*), 301
- setColor() (*psychopy.visual.rift.Rift method*), 328
- setColor() (*psychopy.visual.shape.ShapeStim method*), 349
- setColor() (*psychopy.visual.SphereStim method*), 361
- setColor() (*psychopy.visual.TextBox2 method*), 379
- setConnectionState() (*psychopy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core.EyeTrac method*), 548
- setContrast() (*psychopy.hardware.crs.bits.BitsPlusPlus method*), 474
- setContrast() (*psychopy.hardware.crs.bits.BitsSharp method*), 494
- setContrast() (*psychopy.visual.BoxStim method*), 164
- setContrast() (*psychopy.visual.BufferImageStim method*), 173
- setContrast() (*psychopy.visual.circle.Circle method*), 183
- setContrast() (*psychopy.visual.Form method*), 198
- setContrast() (*psychopy.visual.GratingStim method*), 207
- setContrast() (*psychopy.visual.ImageStim method*), 218
- setContrast() (*psychopy.visual.line.Line method*), 228
- setContrast() (*psychopy.visual.MovieStim method*), 239
- setContrast() (*psychopy.visual.ObjMeshStim method*), 249
- setContrast() (*psychopy.visual.pie.Pie method*), 261
- setContrast() (*psychopy.visual.PlaneStim method*), 269
- setContrast() (*psychopy.visual.polygon.Polygon method*), 278
- setContrast() (*psychopy.visual.RadialStim method*), 288
- setContrast() (*psychopy.visual.rect.Rect method*), 301

- setContrast () (*psychopy.visual.shape.ShapeStim method*), 349
- setContrast () (*psychopy.visual.SphereStim method*), 361
- setContrast () (*psychopy.visual.TextBox2 method*), 379
- setContrast () (*psychopy.visual.TextStim method*), 386
- setCurrent () (*psychopy.monitors.Monitor method*), 763
- setData () (*psychopy.parallel method*), 767
- setDefaultClock () (*in module psychopy.logging*), 756
- setDefaultView () (*psychopy.visual.rift.Rift method*), 329
- setDepth () (*psychopy.visual.BufferImageStim method*), 173
- setDepth () (*psychopy.visual.circle.Circle method*), 183
- setDepth () (*psychopy.visual.Form method*), 198
- setDepth () (*psychopy.visual.GratingStim method*), 208
- setDepth () (*psychopy.visual.ImageStim method*), 218
- setDepth () (*psychopy.visual.line.Line method*), 228
- setDepth () (*psychopy.visual.MovieStim method*), 239
- setDepth () (*psychopy.visual.pie.Pie method*), 262
- setDepth () (*psychopy.visual.polygon.Polygon method*), 278
- setDepth () (*psychopy.visual.RadialStim method*), 288
- setDepth () (*psychopy.visual.rect.Rect method*), 301
- setDepth () (*psychopy.visual.shape.ShapeStim method*), 349
- setDepth () (*psychopy.visual.TextBox2 method*), 379
- setDepth () (*psychopy.visual.TextStim method*), 386
- setDepth () (*psychopy.visual.VlcMovieStim method*), 397
- setDiopters () (*psychopy.visual.nnlvs.VisualSystemHD method*), 415
- setDistance () (*psychopy.monitors.Monitor method*), 764
- setDKL () (*psychopy.visual.BoxStim method*), 164
- setDKL () (*psychopy.visual.BufferImageStim method*), 173
- setDKL () (*psychopy.visual.circle.Circle method*), 183
- setDKL () (*psychopy.visual.Form method*), 198
- setDKL () (*psychopy.visual.GratingStim method*), 207
- setDKL () (*psychopy.visual.ImageStim method*), 218
- setDKL () (*psychopy.visual.line.Line method*), 228
- setDKL () (*psychopy.visual.MovieStim method*), 239
- setDKL () (*psychopy.visual.ObjMeshStim method*), 249
- setDKL () (*psychopy.visual.pie.Pie method*), 261
- setDKL () (*psychopy.visual.PlaneStim method*), 269
- setDKL () (*psychopy.visual.polygon.Polygon method*), 278
- setDKL () (*psychopy.visual.RadialStim method*), 288
- setDKL () (*psychopy.visual.rect.Rect method*), 301
- setDKL () (*psychopy.visual.shape.ShapeStim method*), 349
- setDKL () (*psychopy.visual.SphereStim method*), 361
- setDKL () (*psychopy.visual.TextBox2 method*), 379
- setDKL () (*psychopy.visual.TextStim method*), 386
- setDKL_RGB () (*psychopy.monitors.Monitor method*), 764
- setEdges () (*psychopy.visual.circle.Circle method*), 183
- setEdges () (*psychopy.visual.polygon.Polygon method*), 278
- setEnd () (*psychopy.visual.line.Line method*), 228
- setEnd () (*psychopy.visual.pie.Pie method*), 262
- setExclusive () (*psychopy.event.Mouse method*), 739
- setExp () (*psychopy.data.MultiStairHandler method*), 734
- setExp () (*psychopy.data.PsiHandler method*), 720
- setExp () (*psychopy.data.QuestHandler method*), 725
- setExp () (*psychopy.data.QuestPlusHandler method*), 730
- setExp () (*psychopy.data.StairHandler method*), 716
- setExp () (*psychopy.data.TrialHandler method*), 703
- setExp () (*psychopy.data.TrialHandler2 method*), 708
- setExp () (*psychopy.data.TrialHandlerExt method*), 713
- setEyeOffset () (*psychopy.visual.nnlvs.VisualSystemHD method*), 415
- setFile () (*psychopy.microphone.AdvAudioCapture method*), 758
- setFillColor () (*psychopy.visual.BoxStim method*), 164
- setFillColor () (*psychopy.visual.BufferImageStim method*), 173
- setFillColor () (*psychopy.visual.circle.Circle method*), 183
- setFillColor () (*psychopy.visual.Form method*), 198
- setFillColor () (*psychopy.visual.GratingStim method*), 208
- setFillColor () (*psychopy.visual.ImageStim method*), 218
- setFillColor () (*psychopy.visual.line.Line method*), 229
- setFillColor () (*psychopy.visual.MovieStim method*), 239
- setFillColor () (*psychopy.visual.ObjMeshStim method*), 250
- setFillColor () (*psychopy.visual.pie.Pie method*), 262

setFillColor() (*psychopy.visual.PlaneStim method*), 269
 setFillColor() (*psychopy.visual.polygon.Polygon method*), 278
 setFillColor() (*psychopy.visual.RadialStim method*), 288
 setFillColor() (*psychopy.visual.rect.Rect method*), 301
 setFillColor() (*psychopy.visual.shape.ShapeStim method*), 349
 setFillColor() (*psychopy.visual.SphereStim method*), 361
 setFillColor() (*psychopy.visual.TextBox2 method*), 379
 setFillRGB() (*psychopy.visual.BoxStim method*), 164
 setFillRGB() (*psychopy.visual.BufferImageStim method*), 173
 setFillRGB() (*psychopy.visual.circle.Circle method*), 183
 setFillRGB() (*psychopy.visual.Form method*), 198
 setFillRGB() (*psychopy.visual.GratingStim method*), 208
 setFillRGB() (*psychopy.visual.ImageStim method*), 218
 setFillRGB() (*psychopy.visual.line.Line method*), 229
 setFillRGB() (*psychopy.visual.MovieStim method*), 240
 setFillRGB() (*psychopy.visual.ObjMeshStim method*), 250
 setFillRGB() (*psychopy.visual.pie.Pie method*), 262
 setFillRGB() (*psychopy.visual.PlaneStim method*), 269
 setFillRGB() (*psychopy.visual.polygon.Polygon method*), 278
 setFillRGB() (*psychopy.visual.RadialStim method*), 288
 setFillRGB() (*psychopy.visual.rect.Rect method*), 301
 setFillRGB() (*psychopy.visual.shape.ShapeStim method*), 349
 setFillRGB() (*psychopy.visual.SphereStim method*), 361
 setFillRGB() (*psychopy.visual.TextBox2 method*), 379
 setFlip() (*psychopy.visual.TextStim method*), 387
 setFlipHoriz() (*psychopy.visual.BufferImageStim method*), 173
 setFlipHoriz() (*psychopy.visual.TextStim method*), 387
 setFlipHoriz() (*psychopy.visual.VlcMovieStim method*), 397
 setFlipVert() (*psychopy.visual.BufferImageStim method*), 174
 setFlipVert() (*psychopy.visual.TextStim method*), 387
 setFlipVert() (*psychopy.visual.VlcMovieStim method*), 397
 setFont() (*psychopy.visual.TextBox2 method*), 379
 setFont() (*psychopy.visual.TextStim method*), 387
 setFontColor() (*psychopy.visual.TextBox method*), 370
 setForeColor() (*psychopy.visual.BoxStim method*), 164
 setForeColor() (*psychopy.visual.BufferImageStim method*), 174
 setForeColor() (*psychopy.visual.circle.Circle method*), 183
 setForeColor() (*psychopy.visual.Form method*), 198
 setForeColor() (*psychopy.visual.GratingStim method*), 208
 setForeColor() (*psychopy.visual.ImageStim method*), 218
 setForeColor() (*psychopy.visual.line.Line method*), 229
 setForeColor() (*psychopy.visual.MovieStim method*), 240
 setForeColor() (*psychopy.visual.ObjMeshStim method*), 250
 setForeColor() (*psychopy.visual.pie.Pie method*), 262
 setForeColor() (*psychopy.visual.PlaneStim method*), 269
 setForeColor() (*psychopy.visual.polygon.Polygon method*), 278
 setForeColor() (*psychopy.visual.RadialStim method*), 288
 setForeColor() (*psychopy.visual.rect.Rect method*), 301
 setForeColor() (*psychopy.visual.shape.ShapeStim method*), 349
 setForeColor() (*psychopy.visual.SphereStim method*), 361
 setForeColor() (*psychopy.visual.TextBox2 method*), 379
 setForeColor() (*psychopy.visual.TextStim method*), 387
 setForeRGB() (*psychopy.visual.BoxStim method*), 164
 setForeRGB() (*psychopy.visual.BufferImageStim method*), 174
 setForeRGB() (*psychopy.visual.circle.Circle method*), 183
 setForeRGB() (*psychopy.visual.Form method*), 198
 setForeRGB() (*psychopy.visual.GratingStim method*), 208

setForeRGB () (*psychopy.visual.ImageStim method*), 218
 setForeRGB () (*psychopy.visual.line.Line method*), 229
 setForeRGB () (*psychopy.visual.MovieStim method*), 240
 setForeRGB () (*psychopy.visual.ObjMeshStim method*), 250
 setForeRGB () (*psychopy.visual.pie.Pie method*), 262
 setForeRGB () (*psychopy.visual.PlaneStim method*), 269
 setForeRGB () (*psychopy.visual.polygon.Polygon method*), 278
 setForeRGB () (*psychopy.visual.RadialStim method*), 288
 setForeRGB () (*psychopy.visual.rect.Rect method*), 301
 setForeRGB () (*psychopy.visual.shape.ShapeStim method*), 349
 setForeRGB () (*psychopy.visual.SphereStim method*), 361
 setForeRGB () (*psychopy.visual.TextBox2 method*), 379
 setForeRGB () (*psychopy.visual.TextStim method*), 387
 setGamma () (*psychopy.hardware.crs.bits.BitsPlusPlus method*), 474
 setGamma () (*psychopy.hardware.crs.bits.BitsSharp method*), 494
 setGamma () (*psychopy.monitors.Monitor method*), 764
 setGamma () (*psychopy.visual.nnlvs.VisualSystemHD method*), 415
 setGamma () (*psychopy.visual.rift.Rift method*), 329
 setGammaGrid () (*psychopy.monitors.Monitor method*), 764
 setHeight () (*psychopy.visual.rect.Rect method*), 301
 setHeight () (*psychopy.visual.TextBox2 method*), 379
 setHeight () (*psychopy.visual.TextStim method*), 387
 setHorzAlign () (*psychopy.visual.TextBox method*), 370
 setHorzJust () (*psychopy.visual.TextBox method*), 370
 setIdentity () (*psychopy.visual.RigidBodyPose method*), 339
 setImage () (*psychopy.visual.BufferImageStim method*), 174
 setImage () (*psychopy.visual.ImageStim method*), 218
 setInterpolated () (*psychopy.visual.TextBox method*), 370
 setLevel () (*psychopy.logging.LogFile method*), 754
 setLevelsPost () (*psychopy.monitors.Monitor method*), 764
 setLevelsPre () (*psychopy.monitors.Monitor method*), 764
 setLineariseMethod () (*psychopy.monitors.Monitor method*), 764
 setLineColor () (*psychopy.visual.BoxStim method*), 164
 setLineColor () (*psychopy.visual.BufferImageStim method*), 174
 setLineColor () (*psychopy.visual.Form method*), 198
 setLineColor () (*psychopy.visual.GratingStim method*), 208
 setLineColor () (*psychopy.visual.ImageStim method*), 218
 setLineColor () (*psychopy.visual.MovieStim method*), 240
 setLineColor () (*psychopy.visual.ObjMeshStim method*), 250
 setLineColor () (*psychopy.visual.PlaneStim method*), 269
 setLineColor () (*psychopy.visual.RadialStim method*), 288
 setLineColor () (*psychopy.visual.SphereStim method*), 362
 setLineColor () (*psychopy.visual.TextBox2 method*), 379
 setLineRGB () (*psychopy.visual.BoxStim method*), 164
 setLineRGB () (*psychopy.visual.BufferImageStim method*), 174
 setLineRGB () (*psychopy.visual.circle.Circle method*), 184
 setLineRGB () (*psychopy.visual.Form method*), 198
 setLineRGB () (*psychopy.visual.GratingStim method*), 208
 setLineRGB () (*psychopy.visual.ImageStim method*), 218
 setLineRGB () (*psychopy.visual.line.Line method*), 229
 setLineRGB () (*psychopy.visual.MovieStim method*), 240
 setLineRGB () (*psychopy.visual.ObjMeshStim method*), 250
 setLineRGB () (*psychopy.visual.pie.Pie method*), 262
 setLineRGB () (*psychopy.visual.PlaneStim method*), 269
 setLineRGB () (*psychopy.visual.polygon.Polygon method*), 278
 setLineRGB () (*psychopy.visual.RadialStim method*), 288
 setLineRGB () (*psychopy.visual.rect.Rect method*), 302
 setLineRGB () (*psychopy.visual.shape.ShapeStim method*), 349
 setLineRGB () (*psychopy.visual.SphereStim method*), 362

- setLineRGB () (*psychopy.visual.TextBox2 method*), 379
- setLMS () (*psychopy.visual.BoxStim method*), 164
- setLMS () (*psychopy.visual.BufferImageStim method*), 174
- setLMS () (*psychopy.visual.circle.Circle method*), 183
- setLMS () (*psychopy.visual.Form method*), 198
- setLMS () (*psychopy.visual.GratingStim method*), 208
- setLMS () (*psychopy.visual.ImageStim method*), 218
- setLMS () (*psychopy.visual.line.Line method*), 229
- setLMS () (*psychopy.visual.MovieStim method*), 240
- setLMS () (*psychopy.visual.ObjMeshStim method*), 250
- setLMS () (*psychopy.visual.pie.Pie method*), 262
- setLMS () (*psychopy.visual.PlaneStim method*), 269
- setLMS () (*psychopy.visual.polygon.Polygon method*), 278
- setLMS () (*psychopy.visual.RadialStim method*), 288
- setLMS () (*psychopy.visual.rect.Rect method*), 301
- setLMS () (*psychopy.visual.shape.ShapeStim method*), 349
- setLMS () (*psychopy.visual.SphereStim method*), 362
- setLMS () (*psychopy.visual.TextBox2 method*), 379
- setLMS () (*psychopy.visual.TextStim method*), 387
- setLMS_RGB () (*psychopy.monitors.Monitor method*), 764
- setLumsPost () (*psychopy.monitors.Monitor method*), 764
- setLumsPre () (*psychopy.monitors.Monitor method*), 764
- setLUT () (*psychopy.hardware.crs.bits.BitsPlusPlus method*), 474
- setLUT () (*psychopy.hardware.crs.bits.BitsSharp method*), 494
- setMarker () (*psychopy.microphone.AdvAudioCapture method*), 758
- setMarkerPos () (*psychopy.visual.Slider method*), 355
- setMask () (*psychopy.visual.BufferImageStim method*), 174
- setMask () (*psychopy.visual.GratingStim method*), 208
- setMask () (*psychopy.visual.ImageStim method*), 218
- setMask () (*psychopy.visual.RadialStim method*), 289
- setMaxAttempts () (*psychopy.hardware.minolta.CS100A method*), 516
- setMaxAttempts () (*psychopy.hardware.minolta.LS100 method*), 517
- setMeanLum () (*psychopy.monitors.Monitor method*), 764
- setMode () (*psychopy.hardware.minolta.CS100A method*), 516
- setMode () (*psychopy.hardware.minolta.LS100 method*), 517
- setMouseType () (*psychopy.visual.nnlvs.VisualSystemHD method*), 415
- setMouseType () (*psychopy.visual.rift.Rift method*), 329
- setMouseType () (*psychopy.visual.Window method*), 432
- setMouseVisible () (*psychopy.visual.nnlvs.VisualSystemHD method*), 416
- setMouseVisible () (*psychopy.visual.rift.Rift method*), 329
- setMovie () (*psychopy.visual.MovieStim method*), 240
- setMovie () (*psychopy.visual.VlcMovieStim method*), 397
- setNotes () (*psychopy.monitors.Monitor method*), 764
- setOffAxisView () (*psychopy.visual.nnlvs.VisualSystemHD method*), 416
- setOffAxisView () (*psychopy.visual.rift.Rift method*), 329
- setOffAxisView () (*psychopy.visual.Window method*), 433
- setOpacity () (*psychopy.visual.BufferImageStim method*), 174
- setOpacity () (*psychopy.visual.circle.Circle method*), 184
- setOpacity () (*psychopy.visual.Form method*), 198
- setOpacity () (*psychopy.visual.GratingStim method*), 208
- setOpacity () (*psychopy.visual.ImageStim method*), 218
- setOpacity () (*psychopy.visual.line.Line method*), 229
- setOpacity () (*psychopy.visual.MovieStim method*), 240
- setOpacity () (*psychopy.visual.pie.Pie method*), 262
- setOpacity () (*psychopy.visual.polygon.Polygon method*), 278
- setOpacity () (*psychopy.visual.RadialStim method*), 289
- setOpacity () (*psychopy.visual.rect.Rect method*), 302
- setOpacity () (*psychopy.visual.shape.ShapeStim method*), 350
- setOpacity () (*psychopy.visual.Slider method*), 355
- setOpacity () (*psychopy.visual.TextBox method*), 370
- setOpacity () (*psychopy.visual.TextBox2 method*), 379
- setOpacity () (*psychopy.visual.TextStim method*), 387
- setOpacity () (*psychopy.visual.VlcMovieStim method*), 397
- setOri () (*psychopy.visual.Aperture method*), 157

- setOri () (*psychopy.visual.BoxStim* method), 164
- setOri () (*psychopy.visual.BufferImageStim* method), 174
- setOri () (*psychopy.visual.circle.Circle* method), 184
- setOri () (*psychopy.visual.Form* method), 199
- setOri () (*psychopy.visual.GratingStim* method), 208
- setOri () (*psychopy.visual.ImageStim* method), 218
- setOri () (*psychopy.visual.line.Line* method), 229
- setOri () (*psychopy.visual.MovieStim* method), 240
- setOri () (*psychopy.visual.ObjMeshStim* method), 250
- setOri () (*psychopy.visual.pie.Pie* method), 262
- setOri () (*psychopy.visual.PlaneStim* method), 269
- setOri () (*psychopy.visual.polygon.Polygon* method), 278
- setOri () (*psychopy.visual.RadialStim* method), 289
- setOri () (*psychopy.visual.rect.Rect* method), 302
- setOri () (*psychopy.visual.shape.ShapeStim* method), 350
- setOri () (*psychopy.visual.SphereStim* method), 362
- setOri () (*psychopy.visual.TextBox2* method), 379
- setOri () (*psychopy.visual.TextStim* method), 387
- setOri () (*psychopy.visual.VlcMovieStim* method), 397
- setOriAxisAngle () (*psychopy.visual.BoxStim* method), 164
- setOriAxisAngle () (*psychopy.visual.ObjMeshStim* method), 250
- setOriAxisAngle () (*psychopy.visual.PlaneStim* method), 269
- setOriAxisAngle () (*psychopy.visual.RigidBodyPose* method), 339
- setOriAxisAngle () (*psychopy.visual.SphereStim* method), 362
- setPerspectiveView () (*psychopy.visual.nnlvs.VisualSystemHD* method), 416
- setPerspectiveView () (*psychopy.visual.rift.Rift* method), 330
- setPerspectiveView () (*psychopy.visual.Window* method), 433
- setPhase () (*psychopy.visual.GratingStim* method), 208
- setPhase () (*psychopy.visual.RadialStim* method), 289
- setPin () (*psychopy.parallel* method), 768
- setPortAddress () (*psychopy.parallel* method), 767
- setPos () (*psychopy.event.Mouse* method), 739
- setPos () (*psychopy.visual.Aperture* method), 157
- setPos () (*psychopy.visual.BoxStim* method), 164
- setPos () (*psychopy.visual.BufferImageStim* method), 174
- setPos () (*psychopy.visual.circle.Circle* method), 184
- setPos () (*psychopy.visual.Form* method), 199
- setPos () (*psychopy.visual.GratingStim* method), 208
- setPos () (*psychopy.visual.ImageStim* method), 219
- setPos () (*psychopy.visual.line.Line* method), 229
- setPos () (*psychopy.visual.MovieStim* method), 240
- setPos () (*psychopy.visual.ObjMeshStim* method), 250
- setPos () (*psychopy.visual.pie.Pie* method), 262
- setPos () (*psychopy.visual.PlaneStim* method), 269
- setPos () (*psychopy.visual.polygon.Polygon* method), 278
- setPos () (*psychopy.visual.RadialStim* method), 289
- setPos () (*psychopy.visual.rect.Rect* method), 302
- setPos () (*psychopy.visual.shape.ShapeStim* method), 350
- setPos () (*psychopy.visual.SphereStim* method), 362
- setPos () (*psychopy.visual.TextBox2* method), 379
- setPos () (*psychopy.visual.TextStim* method), 387
- setPos () (*psychopy.visual.VlcMovieStim* method), 397
- setPosition () (*psychopy.visual.TextBox* method), 370
- setPriority () (*psychopy.iohub.client.ioHubConnection* method), 528
- setProcessAffinity () (*psychopy.iohub.client.ioHubConnection* method), 528
- setPsychopyVersion () (*psychopy.monitors.Monitor* method), 764
- setRadialCycles () (*psychopy.visual.RadialStim* method), 289
- setRadialPhase () (*psychopy.visual.RadialStim* method), 289
- setRadius () (*psychopy.visual.circle.Circle* method), 184
- setRadius () (*psychopy.visual.pie.Pie* method), 262
- setRadius () (*psychopy.visual.polygon.Polygon* method), 279
- setReadOnly () (*psychopy.visual.Slider* method), 355
- setRecordFrameIntervals () (*psychopy.visual.nnlvs.VisualSystemHD* method), 416
- setRecordFrameIntervals () (*psychopy.visual.rift.Rift* method), 330
- setRecordingState () (*psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker* method), 538
- setRecordingState () (*psychopy.iohub.devices.eyetracker.hw.mouse.EyeTracker* method), 577
- setRecordingState () (*psychopy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core.EyeTracker* method), 548
- setRecordingState () (*psychopy.iohub.devices.eyetracker.hw.tobii.EyeTracker* method), 570
- setRGB () (*psychopy.visual.BoxStim* method), 164
- setRGB () (*psychopy.visual.BufferImageStim* method), 174

- setRGB () (*psychopy.visual.circle.Circle method*), 184
 setRGB () (*psychopy.visual.Form method*), 199
 setRGB () (*psychopy.visual.GratingStim method*), 208
 setRGB () (*psychopy.visual.ImageStim method*), 219
 setRGB () (*psychopy.visual.line.Line method*), 229
 setRGB () (*psychopy.visual.MovieStim method*), 240
 setRGB () (*psychopy.visual.nnivs.VisualSystemHD method*), 416
 setRGB () (*psychopy.visual.ObjMeshStim method*), 250
 setRGB () (*psychopy.visual.pie.Pie method*), 262
 setRGB () (*psychopy.visual.PlaneStim method*), 269
 setRGB () (*psychopy.visual.polygon.Polygon method*), 278
 setRGB () (*psychopy.visual.RadialStim method*), 289
 setRGB () (*psychopy.visual.rect.Rect method*), 302
 setRGB () (*psychopy.visual.rift.Rift method*), 330
 setRGB () (*psychopy.visual.shape.ShapeStim method*), 350
 setRGB () (*psychopy.visual.SphereStim method*), 362
 setRGB () (*psychopy.visual.TextBox2 method*), 379
 setRGB () (*psychopy.visual.TextStim method*), 387
 setRiftView () (*psychopy.visual.rift.Rift method*), 330
 setRTBoxMode () (*psychopy.hardware.crs.bits.BitsSharp method*), 495
 setScale () (*psychopy.visual.nnivs.VisualSystemHD method*), 416
 setScale () (*psychopy.visual.rift.Rift method*), 330
 setScrollSpeed () (*psychopy.visual.Form method*), 199
 setSF () (*psychopy.visual.GratingStim method*), 208
 setSF () (*psychopy.visual.RadialStim method*), 289
 setSize () (*psychopy.visual.Aperture method*), 157
 setSize () (*psychopy.visual.BufferImageStim method*), 174
 setSize () (*psychopy.visual.circle.Circle method*), 184
 setSize () (*psychopy.visual.Form method*), 199
 setSize () (*psychopy.visual.GratingStim method*), 208
 setSize () (*psychopy.visual.ImageStim method*), 219
 setSize () (*psychopy.visual.line.Line method*), 229
 setSize () (*psychopy.visual.MovieStim method*), 240
 setSize () (*psychopy.visual.pie.Pie method*), 262
 setSize () (*psychopy.visual.polygon.Polygon method*), 279
 setSize () (*psychopy.visual.RadialStim method*), 289
 setSize () (*psychopy.visual.rect.Rect method*), 302
 setSize () (*psychopy.visual.rift.Rift method*), 330
 setSize () (*psychopy.visual.shape.ShapeStim method*), 350
 setSize () (*psychopy.visual.TextBox2 method*), 380
 setSize () (*psychopy.visual.TextStim method*), 387
 setSize () (*psychopy.visual.VlcMovieStim method*), 397
 setSizePix () (*psychopy.monitors.Monitor method*), 764
 setSound () (*psychopy.sound.backend_ptb.SoundPTB method*), 443
 setSound () (*psychopy.sound.backend_sounddevice.SoundDeviceSound method*), 445
 setSpectra () (*psychopy.monitors.Monitor method*), 764
 setStart () (*psychopy.visual.line.Line method*), 229
 setStart () (*psychopy.visual.pie.Pie method*), 262
 setStatusBoxMode () (*psychopy.hardware.crs.bits.BitsSharp method*), 495
 setStatusBoxThreshold () (*psychopy.hardware.crs.bits.BitsSharp method*), 495
 setStatusEventParams () (*psychopy.hardware.crs.bits.BitsSharp method*), 495
 setStereoDebugHudOption () (*psychopy.visual.rift.Rift method*), 330
 setTex () (*psychopy.visual.GratingStim method*), 208
 setTex () (*psychopy.visual.RadialStim method*), 289
 setText () (*psychopy.visual.TextBox method*), 371
 setText () (*psychopy.visual.TextBox2 method*), 380
 setText () (*psychopy.visual.TextStim method*), 387
 setTextGridLineColor () (*psychopy.visual.TextBox method*), 371
 setTextGridLineWidth () (*psychopy.visual.TextBox method*), 371
 setToeInView () (*psychopy.visual.nnivs.VisualSystemHD method*), 416
 setToeInView () (*psychopy.visual.rift.Rift method*), 331
 setToeInView () (*psychopy.visual.Window method*), 433
 setTrigger () (*psychopy.hardware.crs.bits.BitsPlusPlus method*), 474
 setTrigger () (*psychopy.hardware.crs.bits.BitsSharp method*), 496
 setTriggerList () (*psychopy.hardware.crs.bits.BitsPlusPlus method*), 475
 setTriggerList () (*psychopy.hardware.crs.bits.BitsSharp method*), 496
 setUnits () (*psychopy.visual.nnivs.VisualSystemHD method*), 417
 setUnits () (*psychopy.visual.rift.Rift method*), 331
 setupProxy () (*in module psychopy.web*), 777
 setUseBits () (*psychopy.monitors.Monitor method*), 764

setVertAlign() (*psychopy.visual.TextBox* method), 371
 setVertexAttribPointer() (in module *psychopy.tools.gltools*), 616
 setVertices() (*psychopy.visual.circle.Circle* method), 184
 setVertices() (*psychopy.visual.line.Line* method), 229
 setVertices() (*psychopy.visual.pie.Pie* method), 262
 setVertices() (*psychopy.visual.polygon.Polygon* method), 279
 setVertices() (*psychopy.visual.rect.Rect* method), 302
 setVertices() (*psychopy.visual.shape.ShapeStim* method), 350
 setVertJust() (*psychopy.visual.TextBox* method), 371
 setViewPos() (*psychopy.visual.nnlvs.VisualSystemHD* method), 417
 setViewPos() (*psychopy.visual.rift.Rift* method), 331
 setVisible() (*psychopy.event.Mouse* method), 739
 setVolume() (*psychopy.sound.backend_pygame.SoundPygame* method), 448
 setVolume() (*psychopy.visual.VlcMovieStim* method), 398
 setWidth() (*psychopy.monitors.Monitor* method), 764
 setWidth() (*psychopy.visual.rect.Rect* method), 302
 sf (*psychopy.visual.GratingStim* attribute), 208
 sf (*psychopy.visual.RadialStim* attribute), 289
 ShapeStim (class in *psychopy.visual.shape*), 342
 shininess() (*psychopy.visual.BlinnPhongMaterial* property), 253
 shouldQuit() (*psychopy.visual.rift.Rift* property), 331
 shouldRecenter() (*psychopy.visual.rift.Rift* property), 331
 show() (*psychopy.gui.Dlg* method), 745
 show() (*psychopy.gui.DlgFromDict* method), 745
 shutdown() (*psychopy.iohub.client.ioHubConnection* method), 529
 signalDots (*psychopy.visual.DotStim* attribute), 186
 silence() (*psychopy.sound.AudioClip* static method), 456
 SimpleImageStim (class in *psychopy.visual*), 351
 simulate() (*psychopy.data.QuestHandler* method), 726
 sine() (*psychopy.sound.AudioClip* static method), 457
 Size (class in *psychopy.layout*), 751
 size (*psychopy.visual.Window* attribute), 421
 size() (*psychopy.layout.Vertices* property), 753
 size() (*psychopy.visual.Aperture* property), 157
 size() (*psychopy.visual.BoxStim* property), 164
 size() (*psychopy.visual.BufferImageStim* property), 174
 size() (*psychopy.visual.circle.Circle* property), 184
 size() (*psychopy.visual.Form* property), 199
 size() (*psychopy.visual.GratingStim* property), 209
 size() (*psychopy.visual.ImageStim* property), 219
 size() (*psychopy.visual.line.Line* property), 229
 size() (*psychopy.visual.MovieStim* property), 240
 size() (*psychopy.visual.nnlvs.VisualSystemHD* property), 417
 size() (*psychopy.visual.ObjMeshStim* property), 250
 size() (*psychopy.visual.pie.Pie* property), 262
 size() (*psychopy.visual.PlaneStim* property), 269
 size() (*psychopy.visual.polygon.Polygon* property), 279
 size() (*psychopy.visual.RadialStim* property), 289
 size() (*psychopy.visual.rect.Rect* property), 302
 size() (*psychopy.visual.rift.Rift* property), 331
 size() (*psychopy.visual.shape.ShapeStim* property), 350
 size() (*psychopy.visual.Slider* property), 355
 size() (*psychopy.visual.SphereStim* property), 362
 size() (*psychopy.visual.TextBox2* property), 380
 size() (*psychopy.visual.TextStim* property), 387
 size() (*psychopy.visual.VlcMovieStim* property), 398
 size() (*psychopy.visual.Window* property), 434
 sizePix() (*psychopy.visual.Aperture* property), 158
 skyCubeMap() (*psychopy.visual.SceneSkybox* property), 340
 slerp() (in module *psychopy.tools.mathtools*), 650
 Slider (class in *psychopy.visual*), 352
 slippage() (*psychopy.voicekey.OnsetVoiceKey* property), 775
 smooth() (in module *psychopy.voicekey*), 775
 SoundDeviceSound (class in *psychopy.sound.backend_sounddevice*), 444
 SoundPTB (class in *psychopy.sound.backend_ptb*), 442
 SoundPygame (class in *psychopy.sound.backend_pygame*), 447
 SoundPyo (class in *psychopy.sound.backend_pyo*), 446
 specifyTrackingOrigin() (*psychopy.visual.rift.Rift* method), 331
 specifyTrackingOriginPosOri() (*psychopy.visual.rift.Rift* method), 331
 specularColor() (*psychopy.visual.BlinnPhongMaterial* property), 253
 specularColor() (*psychopy.visual.LightSource* property), 221
 specularRGB() (*psychopy.visual.BlinnPhongMaterial* property), 253
 specularRGB() (*psychopy.visual.LightSource* property), 221

speed (*psychopy.visual.DotStim* attribute), 187
 sph2cart () (in module *psy-
chopy.tools.coordinatestools*), 587
 SphereStim (class in *psychopy.visual*), 356
 square () (*psychopy.sound.AudioClip* static method),
457
 srgb () (*psychopy.colors.Color* property), 695
 srgbTF () (in module *psychopy.tools.colorspectools*),
585
 StairHandler (class in *psychopy.data*), 713
 start (*psychopy.visual.line.Line* attribute), 230
 start (*psychopy.visual.pie.Pie* attribute), 263
 start () (*psychopy.clock.StaticPeriod* method), 154
 start () (*psychopy.core.StaticPeriod* method), 151
 start () (*psychopy.hardware.crs.bits.BitsSharp*
method), 497
 start () (*psychopy.hardware.emulator.ResponseEmulator*
method), 506
 start () (*psychopy.hardware.emulator.SyncGenerator*
method), 508
 start () (*psychopy.hardware.keyboard.Keyboard*
method), 466
 start () (*psychopy.sound.Microphone* method), 451
 start () (*psychopy.voicekey.OnsetVoiceKey* method),
775
 start_angle_x (psy-
chopy.iohub.devices.eyetracker.FixationEndEvent
attribute), 558
 start_angle_x (psy-
chopy.iohub.devices.eyetracker.SaccadeEndEvent
attribute), 560
 start_angle_y (psy-
chopy.iohub.devices.eyetracker.FixationEndEvent
attribute), 558
 start_angle_y (psy-
chopy.iohub.devices.eyetracker.SaccadeEndEvent
attribute), 560
 start_gaze_x (psy-
chopy.iohub.devices.eyetracker.FixationEndEvent
attribute), 558, 579
 start_gaze_x (psy-
chopy.iohub.devices.eyetracker.SaccadeEndEvent
attribute), 560
 start_gaze_y (psy-
chopy.iohub.devices.eyetracker.FixationEndEvent
attribute), 558, 579
 start_gaze_y (psy-
chopy.iohub.devices.eyetracker.SaccadeEndEvent
attribute), 560
 start_ppd_x (*psychopy.iohub.devices.eyetracker.FixationEndEvent*
attribute), 558
 start_ppd_x (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent*
attribute), 560
 start_ppd_y (*psychopy.iohub.devices.eyetracker.FixationEndEvent*
attribute), 558
 start_ppd_y (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent*
attribute), 560
 start_pupil_measure1_type (psy-
chopy.iohub.devices.eyetracker.FixationEndEvent
attribute), 558
 start_pupil_measure1_type (psy-
chopy.iohub.devices.eyetracker.SaccadeEndEvent
attribute), 560
 start_pupil_measure_1 (psy-
chopy.iohub.devices.eyetracker.FixationEndEvent
attribute), 558
 start_pupil_measure_1 (psy-
chopy.iohub.devices.eyetracker.SaccadeEndEvent
attribute), 560
 start_velocity_xy (psy-
chopy.iohub.devices.eyetracker.FixationEndEvent
attribute), 558
 start_velocity_xy (psy-
chopy.iohub.devices.eyetracker.SaccadeEndEvent
attribute), 560
 startAnalog () (psy-
chopy.hardware.crs.bits.BitsSharp method),
497
 startApp () (in module *psychopy.app*), 693
 startCustomTasklet () (psy-
chopy.iohub.client.ioHubConnection method),
528
 started () (*psychopy.voicekey.OnsetVoiceKey* prop-
erty), 775
 startGoggles () (psy-
chopy.hardware.crs.bits.BitsPlusPlus method),
475
 startGoggles () (psy-
chopy.hardware.crs.bits.BitsSharp method),
497
 startHaptics () (*psychopy.visual.rift.Rift* method),
332
 startIntensity () (psy-
chopy.data.QuestPlusHandler property),
730
 startOfFlip () (psy-
chopy.visual.windowframepack.ProjectorFramePacker
method), 436
 startRecording () (psy-
chopy.hardware.brainproducts.RemoteControlServer
method), 469
 startRemoteMode () (*psychopy.hardware.pr.PR655*
method), 520
 startStatusLog () (psy-
chopy.hardware.crs.bits.BitsSharp method),
498
 startText () (*psychopy.visual.TextBox2* property),
680

startTime() (*psychopy.sound.AudioDeviceStatus* property), 464
 startTrigger() (*psychopy.hardware.crs.bits.BitsPlusPlus* method), 476
 startTrigger() (*psychopy.hardware.crs.bits.BitsSharp* method), 498
 startUpPlugins() (*in module psychopy.plugins*), 770
 state() (*psychopy.iohub.client.keyboard.Keyboard* property), 532
 state() (*psychopy.sound.AudioDeviceStatus* property), 464
 StaticPeriod (*class in psychopy.clock*), 153
 StaticPeriod (*class in psychopy.core*), 150
 status (*psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent* attribute), 540, 556, 571, 578
 status (*psychopy.iohub.devices.eyetracker.BlinkEndEvent* attribute), 561
 status (*psychopy.iohub.devices.eyetracker.BlinkStartEvent* attribute), 561
 status (*psychopy.iohub.devices.eyetracker.FixationEndEvent* attribute), 559
 status (*psychopy.iohub.devices.eyetracker.FixationStartEvent* attribute), 557
 status (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent* attribute), 555
 status (*psychopy.iohub.devices.eyetracker.SaccadeEndEvent* attribute), 561
 status (*psychopy.iohub.devices.eyetracker.SaccadeStartEvent* attribute), 559
 status() (*psychopy.sound.backend_ptb.SoundPTB* property), 444
 status() (*psychopy.sound.Microphone* property), 452
 statusBoxAddKeys() (*psychopy.hardware.crs.bits.BitsSharp* method), 498
 statusBoxDisable() (*psychopy.hardware.crs.bits.BitsSharp* method), 499
 statusBoxEnable() (*psychopy.hardware.crs.bits.BitsSharp* method), 499
 statusBoxKeysPressed() (*psychopy.hardware.crs.bits.BitsSharp* method), 500
 statusBoxResetKeys() (*psychopy.hardware.crs.bits.BitsSharp* method), 500
 statusBoxSetKeys() (*psychopy.hardware.crs.bits.BitsSharp* method), 500
 statusBoxWait() (*psychopy.hardware.crs.bits.BitsSharp* method), 501
 statusBoxWaitN() (*psychopy.hardware.crs.bits.BitsSharp* method), 501
 std() (*in module psychopy.voicekey*), 775
 stencilTest() (*psychopy.visual.nnivs.VisualSystemHD* property), 417
 stencilTest() (*psychopy.visual.rift.Rift* property), 332
 stencilTest() (*psychopy.visual.Window* property), 434
 stereoDebugHudMode() (*psychopy.visual.rift.Rift* method), 332
 stop() (*psychopy.hardware.crs.bits.BitsSharp* method), 502
 stop() (*psychopy.hardware.keyboard.Keyboard* method), 466
 stop() (*psychopy.hardware.qmix.Pump* method), 522
 stop() (*psychopy.microphone.AdvAudioCapture* method), 758
 stop() (*psychopy.sound.backend_ptb.SoundPTB* method), 444
 stop() (*psychopy.sound.backend_pygame.SoundPygame* method), 448
 stop() (*psychopy.sound.backend_pyo.SoundPyo* method), 446
 stop() (*psychopy.sound.backend_sounddevice.SoundDeviceSound* method), 445
 stop() (*psychopy.sound.Microphone* method), 452
 stop() (*psychopy.visual.MovieStim* method), 240
 stop() (*psychopy.visual.VlcMovieStim* method), 398
 stop() (*psychopy.voicekey.OnsetVoiceKey* method), 775
 stopAnalog() (*psychopy.hardware.crs.bits.BitsSharp* method), 502
 stopCustomTasklet() (*psychopy.iohub.client.ioHubConnection* method), 529
 stopGoggles() (*psychopy.hardware.crs.bits.BitsPlusPlus* method), 476
 stopGoggles() (*psychopy.hardware.crs.bits.BitsSharp* method), 502
 stopHaptics() (*psychopy.visual.rift.Rift* method), 332
 stopRecording() (*psychopy.hardware.brainproducts.RemoteControlServer* method), 469
 stopStatusLog() (*psychopy.hardware.crs.bits.BitsSharp* method), 502

stopTrigger() (psychopy.hardware.crs.bits.BitsPlusPlus method), 476
 stopTrigger() (psychopy.hardware.crs.bits.BitsSharp method), 503
 stream() (psychopy.sound.backend_ptb.SoundPTB property), 444
 stream() (psychopy.sound.backend_sounddevice.SoundDevice property), 445
 streamBufferSecs() (psychopy.sound.Microphone property), 452
 streamStatus() (psychopy.sound.Microphone property), 452
 style() (psychopy.visual.Form property), 199
 style() (psychopy.visual.Slider property), 355
 styleTweaks (psychopy.visual.Slider attribute), 355
 submitControllerVibration() (psychopy.visual.rift.Rift method), 332
 surface_topic() (psychopy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core.EyeTracker property), 548
 surfaceBitangent() (in module psychopy.tools.mathtools), 643
 surfaceNormal() (in module psychopy.tools.mathtools), 642
 surfaceTangent() (in module psychopy.tools.mathtools), 644
 switchOn() (in module psychopy.microphone), 757
 switchValvePosition() (psychopy.hardware.qmix.Pump method), 522
 syncClocks() (psychopy.hardware.crs.bits.BitsPlusPlus method), 476
 syncClocks() (psychopy.hardware.crs.bits.BitsSharp method), 503
 SyncGenerator (class in psychopy.hardware.emulator), 506, 509
 syringeType() (psychopy.hardware.qmix.Pump property), 522

T

table_from_file() (in module psychopy.voicekey), 776
 table_from_samples() (in module psychopy.voicekey), 776
 table_to_file() (in module psychopy.voicekey), 776
 tanAngleToNDC() (psychopy.visual.rift.Rift method), 333
 temporalDithering() (psychopy.hardware.crs.bits.BitsSharp property), 503
 testBoundary() (psychopy.visual.rift.Rift method), 333
 tex (psychopy.visual.GratingStim attribute), 209
 tex (psychopy.visual.RadialStim attribute), 290
 texRes (psychopy.visual.BufferImageStim attribute), 175
 texRes (psychopy.visual.GratingStim attribute), 209
 texRes (psychopy.visual.ImageStim attribute), 219
 texRes (psychopy.visual.RadialStim attribute), 290
 text (psychopy.visual.TextStim attribute), 388
 text() (psychopy.visual.TextBox2 property), 380
 TextBox (class in psychopy.visual), 365
 TextBox2 (class in psychopy.visual), 372
 TextStim (class in psychopy.visual), 381
 thePose() (psychopy.visual.BoxStim property), 164
 thePose() (psychopy.visual.ObjMeshStim property), 250
 thePose() (psychopy.visual.PlaneStim property), 269
 thePose() (psychopy.visual.SphereStim property), 362
 time (psychopy.iohub.devices.eyetracker.BinocularEyeSampleEvent attribute), 539, 550, 555, 570, 578
 time (psychopy.iohub.devices.eyetracker.BlinkEndEvent attribute), 561
 time (psychopy.iohub.devices.eyetracker.BlinkStartEvent attribute), 561
 time (psychopy.iohub.devices.eyetracker.FixationEndEvent attribute), 540, 557, 578
 time (psychopy.iohub.devices.eyetracker.FixationStartEvent attribute), 540, 557, 578
 time (psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent attribute), 549, 554
 time (psychopy.iohub.devices.eyetracker.SaccadeEndEvent attribute), 560
 time (psychopy.iohub.devices.eyetracker.SaccadeStartEvent attribute), 559
 time() (psychopy.iohub.client.keyboard.KeyboardPress property), 533
 time() (psychopy.iohub.client.keyboard.KeyboardRelease property), 534
 timeFailed() (psychopy.sound.AudioDeviceStatus property), 464
 timeOnFlip() (psychopy.visual.nnlvs.VisualSystemHD method), 417
 timeOnFlip() (psychopy.visual.rift.Rift method), 333
 timeOnFlip() (psychopy.visual.Window method), 434
 timeout() (psychopy.hardware.brainproducts.RemoteControlServer property), 469
 timestampOnFlip() (psychopy.data.ExperimentHandler method), 698
 toFile() (in module psychopy.tools.filetools), 587
 tone() (in module psychopy.voicekey), 776
 totalCalls() (psychopy.sound.AudioDeviceStatus

property), 464
 track () (*psychopy.sound.backend_ptb.SoundPTB property*), 444
 trackerCount () (*psychopy.visual.rift.Rift property*), 333
 trackerSec () (*psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker method*), 538
 trackerSec () (*psychopy.iohub.devices.eyetracker.hw.mouse.EyeTracker method*), 577
 trackerSec () (*psychopy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core.EyeTracker method*), 549
 trackerTime () (*psychopy.iohub.devices.eyetracker.hw.gazepoint.gp3.EyeTracker method*), 538
 trackerTime () (*psychopy.iohub.devices.eyetracker.hw.mouse.EyeTracker method*), 577
 trackerTime () (*psychopy.iohub.devices.eyetracker.hw.pupil_labs.pupil_core.EyeTracker method*), 549
 trackingOriginType () (*psychopy.visual.rift.Rift property*), 333
 transcribe () (*psychopy.sound.AudioClip method*), 458
 transform () (*in module psychopy.tools.mathtools*), 647
 transform () (*psychopy.visual.RigidBodyPose method*), 339
 transformMeshPosOri () (*in module psychopy.tools.gltools*), 630
 transformNormal () (*psychopy.visual.RigidBodyPose method*), 339
 translationMatrix () (*in module psychopy.tools.mathtools*), 662
 TrialHandler (*class in psychopy.data*), 699
 TrialHandler2 (*class in psychopy.data*), 704
 TrialHandlerExt (*class in psychopy.data*), 708
 type () (*psychopy.iohub.client.keyboard.KeyboardPress property*), 533
 type () (*psychopy.iohub.client.keyboard.KeyboardRelease property*), 534

U

uint8_float () (*in module psychopy.tools.typetools*), 680
 unbindTexture () (*in module psychopy.tools.gltools*), 608
 unbindVBO () (*in module psychopy.tools.gltools*), 615
 uncompress () (*psychopy.microphone.AdvAudioCapture method*), 758
 units (*psychopy.visual.BoxStim attribute*), 164
 units (*psychopy.visual.nnlvs.VisualSystemHD attribute*), 417
 units (*psychopy.visual.ObjMeshStim attribute*), 250
 units (*psychopy.visual.PlaneStim attribute*), 269
 units (*psychopy.visual.rift.Rift attribute*), 333
 units (*psychopy.visual.SphereStim attribute*), 362
 units (*psychopy.visual.Window attribute*), 434
 units () (*psychopy.event.Mouse property*), 739
 units () (*psychopy.layout.Vertices property*), 753
 units () (*psychopy.visual.BufferImageStim property*), 175
 units (*psychopy.visual.Form property*), 199
 units () (*psychopy.visual.GratingStim property*), 209
 units () (*psychopy.visual.ImageStim property*), 219
 units (*psychopy.visual.MovieStim property*), 241
 units () (*psychopy.visual.RadialStim property*), 290
 units () (*psychopy.visual.Slider property*), 356
 units () (*psychopy.visual.TextBox2 property*), 380
 units () (*psychopy.visual.VlcMovieStim property*), 398
 unmapBuffer () (*in module psychopy.tools.gltools*), 616
 up () (*psychopy.visual.RigidBodyPose property*), 339
 update () (*psychopy.visual.nnlvs.VisualSystemHD method*), 417
 update () (*psychopy.visual.rift.Rift method*), 334
 updateColors () (*psychopy.visual.BoxStim method*), 165
 updateColors () (*psychopy.visual.BufferImageStim method*), 175
 updateColors () (*psychopy.visual.circle.Circle method*), 184
 updateColors () (*psychopy.visual.Form method*), 199
 updateColors () (*psychopy.visual.GratingStim method*), 209
 updateColors () (*psychopy.visual.ImageStim method*), 219
 updateColors () (*psychopy.visual.line.Line method*), 230
 updateColors () (*psychopy.visual.MovieStim method*), 241
 updateColors () (*psychopy.visual.ObjMeshStim method*), 250
 updateColors () (*psychopy.visual.pie.Pie method*), 263
 updateColors () (*psychopy.visual.PlaneStim method*), 270
 updateColors () (*psychopy.visual.polygon.Polygon method*), 279
 updateColors () (*psychopy.visual.RadialStim method*), 290
 updateColors () (*psychopy.visual.rect.Rect method*), 302

- updateColors() (*psychopy.visual.shape.ShapeStim method*), 350
 - updateColors() (*psychopy.visual.SphereStim method*), 362
 - updateColors() (*psychopy.visual.TextBox2 method*), 380
 - updateColors() (*psychopy.visual.TextStim method*), 388
 - updateInputState() (*psychopy.visual.rift.Rift method*), 334
 - updateLights() (*psychopy.visual.nnlvs.VisualSystemHD method*), 417
 - updateLights() (*psychopy.visual.rift.Rift method*), 334
 - updateLights() (*psychopy.visual.Window method*), 434
 - updateOpacity() (*psychopy.visual.BufferImageStim method*), 175
 - updateOpacity() (*psychopy.visual.circle.Circle method*), 184
 - updateOpacity() (*psychopy.visual.Form method*), 199
 - updateOpacity() (*psychopy.visual.GratingStim method*), 209
 - updateOpacity() (*psychopy.visual.ImageStim method*), 219
 - updateOpacity() (*psychopy.visual.line.Line method*), 230
 - updateOpacity() (*psychopy.visual.MovieStim method*), 241
 - updateOpacity() (*psychopy.visual.pie.Pie method*), 263
 - updateOpacity() (*psychopy.visual.polygon.Polygon method*), 279
 - updateOpacity() (*psychopy.visual.RadialStim method*), 290
 - updateOpacity() (*psychopy.visual.rect.Rect method*), 302
 - updateOpacity() (*psychopy.visual.shape.ShapeStim method*), 350
 - updateOpacity() (*psychopy.visual.TextBox2 method*), 380
 - updateOpacity() (*psychopy.visual.TextStim method*), 388
 - updateOpacity() (*psychopy.visual.VlcMovieStim method*), 398
 - updateTexture() (*psychopy.visual.VlcMovieStim method*), 398
 - updateVideoFrame() (*psychopy.visual.MovieStim method*), 241
 - useFBO() (*in module psychopy.tools.gltools*), 603
 - useLights() (*in module psychopy.tools.gltools*), 621
 - useLights() (*psychopy.visual.nnlvs.VisualSystemHD property*), 418
 - useLights() (*psychopy.visual.rift.Rift property*), 334
 - useLights() (*psychopy.visual.Window property*), 434
 - useMaterial() (*in module psychopy.tools.gltools*), 620
 - useProgram() (*in module psychopy.tools.gltools*), 595
 - useProgramObjectARB() (*in module psychopy.tools.gltools*), 596
 - userData() (*psychopy.sound.AudioClip property*), 458
 - userHeight() (*psychopy.visual.rift.Rift property*), 334
- ## V
- validate() (*psychopy.colors.Color method*), 695
 - validate() (*psychopy.layout.Position method*), 751
 - validate() (*psychopy.layout.Size method*), 752
 - validate() (*psychopy.layout.Vector method*), 750
 - validate() (*psychopy.preferences.Preferences method*), 772
 - validateProgram() (*in module psychopy.tools.gltools*), 595
 - validateProgramARB() (*in module psychopy.tools.gltools*), 595
 - value() (*psychopy.visual.Slider property*), 356
 - values() (*psychopy.visual.Form property*), 199
 - VBI, 30
 - VBI blocking, 30
 - VBI syncing, 30
 - Vector (*class in psychopy.layout*), 749
 - velocity_x (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent attribute*), 555
 - velocity_xy (*psychopy.iohub.devices.eyetracker.FixationStartEvent attribute*), 557
 - velocity_xy (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent attribute*), 555
 - velocity_xy (*psychopy.iohub.devices.eyetracker.SaccadeStartEvent attribute*), 559
 - velocity_y (*psychopy.iohub.devices.eyetracker.MonocularEyeSampleEvent attribute*), 555
 - version() (*psychopy.hardware.brainproducts.RemoteControlServer property*), 469
 - VertexArrayInfo (*class in psychopy.tools.gltools*), 609
 - VertexBufferInfo (*class in psychopy.tools.gltools*), 612
 - vertexNormal() (*in module psychopy.tools.mathtools*), 645
 - Vertices (*class in psychopy.layout*), 752
 - vertices() (*psychopy.visual.Aperture property*), 158
 - vertices() (*psychopy.visual.BoxStim property*), 165
 - vertices() (*psychopy.visual.BufferImageStim property*), 175
 - vertices() (*psychopy.visual.Form property*), 199

- vertices () (*psychopy.visual.GratingStim* property), 209
 - vertices () (*psychopy.visual.ImageStim* property), 219
 - vertices () (*psychopy.visual.line.Line* property), 230
 - vertices () (*psychopy.visual.MovieStim* property), 241
 - vertices () (*psychopy.visual.ObjMeshStim* property), 251
 - vertices () (*psychopy.visual.PlaneStim* property), 270
 - vertices () (*psychopy.visual.RadialStim* property), 290
 - vertices () (*psychopy.visual.shape.ShapeStim* property), 350
 - vertices () (*psychopy.visual.SphereStim* property), 362
 - vertices () (*psychopy.visual.TextBox2* property), 380
 - vertices () (*psychopy.visual.VlcMovieStim* property), 398
 - verticesPix () (*psychopy.visual.Aperture* property), 158
 - verticesPix () (*psychopy.visual.BufferImageStim* property), 175
 - verticesPix () (*psychopy.visual.circle.Circle* property), 184
 - verticesPix () (*psychopy.visual.Form* property), 199
 - verticesPix () (*psychopy.visual.GratingStim* property), 209
 - verticesPix () (*psychopy.visual.ImageStim* property), 219
 - verticesPix () (*psychopy.visual.line.Line* property), 230
 - verticesPix () (*psychopy.visual.MovieStim* property), 241
 - verticesPix () (*psychopy.visual.pie.Pie* property), 263
 - verticesPix () (*psychopy.visual.polygon.Polygon* property), 279
 - verticesPix () (*psychopy.visual.RadialStim* property), 290
 - verticesPix () (*psychopy.visual.rect.Rect* property), 302
 - verticesPix () (*psychopy.visual.shape.ShapeStim* property), 350
 - verticesPix () (*psychopy.visual.TextBox2* property), 380
 - verticesPix () (*psychopy.visual.TextStim* property), 388
 - verticesPix () (*psychopy.visual.VlcMovieStim* property), 398
 - videoSize () (*psychopy.visual.MovieStim* property), 241
 - videoSize () (*psychopy.visual.VlcMovieStim* property), 398
 - viewMatrix () (*psychopy.visual.nnlvs.VisualSystemHD* property), 418
 - viewMatrix () (*psychopy.visual.rift.Rift* property), 334
 - viewMatrix () (*psychopy.visual.Window* property), 435
 - viewport () (*psychopy.visual.nnlvs.VisualSystemHD* property), 418
 - viewport () (*psychopy.visual.rift.Rift* property), 335
 - viewport () (*psychopy.visual.Window* property), 435
 - viewPos (*psychopy.visual.nnlvs.VisualSystemHD* attribute), 418
 - viewPos (*psychopy.visual.rift.Rift* attribute), 334
 - viewPos (*psychopy.visual.Window* attribute), 435
 - visible () (*in module psychopy.tools.viewtools*), 691
 - visibleBBox () (*in module psychopy.tools.viewtools*), 692
 - visibleText () (*psychopy.visual.TextBox2* property), 380
 - visibleWedge (*psychopy.visual.RadialStim* attribute), 290
 - visualAngle () (*in module psychopy.tools.viewtools*), 682
 - VisualSystemHD (*class in psychopy.visual.nnlvs*), 399
 - VlcMovieStim (*class in psychopy.visual*), 390
 - volume () (*psychopy.visual.MovieStim* property), 241
 - volume () (*psychopy.visual.VlcMovieStim* property), 398
 - volumeDown () (*psychopy.visual.MovieStim* method), 241
 - volumeUnit () (*psychopy.hardware.qmix.Pump* property), 522
 - volumeUp () (*psychopy.visual.MovieStim* method), 241
- ## W
- wait () (*in module psychopy.clock*), 154
 - wait () (*in module psychopy.core*), 151
 - wait_for_event () (*psychopy.voicekey.OnsetVoiceKey* method), 775
 - waitBlanking (*psychopy.visual.nnlvs.VisualSystemHD* attribute), 418
 - waitBlanking (*psychopy.visual.rift.Rift* attribute), 335
 - waitBlanking (*psychopy.visual.Window* attribute), 435
 - waitForKeys () (*psychopy.iohub.client.keyboard.Keyboard* method), 532

waitForMessage () (psy- win () (*psychopy.visual.VlcMovieStim* property), 398
chopy.hardware.brainproducts.RemoteControlServer
method), 469 window (class in *psychopy.visual*), 419
waitForPresses () (psy- windowedSize () (psy-
chopy.iohub.client.keyboard.Keyboard *chopy.visual.nnivs.VisualSystemHD* property),
method), 532 418
waitForReleases () (psy- windowedSize () (*psychopy.visual.rift.Rift* property),
chopy.iohub.client.keyboard.Keyboard 335
method), 532 windowedSize () (*psychopy.visual.Window* property),
435
waitForState () (psy- workspace () (*psychopy.hardware.brainproducts.RemoteControlServer*
chopy.hardware.brainproducts.RemoteControlServer property), 470
method), 470 wrapWidth (*psychopy.visual.TextStim* attribute), 388
write () (*psychopy.logging.LogFile* method), 754

X

XboxController (class in psy-
chopy.hardware.joystick), 511

xlsx, 30

xRuns () (*psychopy.sound.AudioDeviceStatus* prop-
erty), 464

xydist () (in module *psychopy.event*), 740

Z

zero_crossings () (in module *psychopy.voicekey*),
775

zeroFix () (in module *psychopy.tools.mathtools*), 676