
pyglet Documentation

Release 1.2.2

Alex Holkner

February 21, 2017

1	Programming Guide	3
1.1	pyglet Programming Guide	3
2	API Reference	89
2.1	pyglet	89
3	Development guide	317
3.1	Development environment	317
3.2	Testing pyglet	320
3.3	Documentation	322
3.4	Making a pyglet release	328
3.5	OpenGL Interface Implementation	329
3.6	ctypes Wrapper Generation	329
3.7	wraptypes	331
4	Related Documentation	337
	Python Module Index	339

Pyglet is a pure python cross-platform application framework intended for game development. It supports windowing, user interface event handling, OpenGL graphics, loading images and videos and playing sounds and music. It works on Windows, OS X and Linux.

Programming Guide

Every programmer starting to develop using pyglet, should start here.

pyglet Programming Guide

The pyglet Programming Guide provides in-depth documentation for writing applications that use pyglet. Many topics described here reference the pyglet API reference, provided separately.

If this is your first time reading about pyglet, we suggest you start at *Writing a pyglet application*.

Installation

pyglet does not need to be installed. Because it uses no external libraries or compiled binaries, you can run it in-place. You can distribute the pyglet source code or runtime eggs alongside your application code (see *Distribution*).

You might want to experiment with pyglet and run the example programs before you install it on your development machine. To do this, add either the extracted pyglet source archive directory or the compressed runtime egg to your PYTHONPATH.

On Windows you can specify this from a command line:

```
set PYTHONPATH c:\path\to\pyglet-1.1\;%PYTHONPATH%
```

On Mac OS X, Linux or on Windows under cygwin using bash:

```
set PYTHONPATH /path/to/pyglet-1.1/:$PYTHONPATH
export PYTHONPATH
```

or, using tcsh or a variant:

```
setenv PYTHONPATH /path/to/pyglet-1.1/:$PYTHONPATH
```

If you have downloaded a runtime egg instead of the source archive, you would specify the filename of the egg in place of pyglet-1.1/.

- *Installing using setup.py*
- *Installation from the runtime eggs*

Installing using setup.py

To make pyglet available to all users, or to avoid having to set the `PYTHONPATH` for each session, you can install it into your Python's `site-packages` directory.

From a command prompt on Windows, change into the extracted pyglet source archive directory and type:

```
python setup.py install
```

On Mac OS X and Linux you will need to do the above as a privileged user; for example using `sudo`:

```
sudo python setup.py install
```

Once installed you should be able to `import pyglet` from any terminal without setting the `PYTHONPATH`.

Installation from the runtime eggs

If you have *setuptools* installed, you can install or upgrade to the latest version of pyglet using `easy_install`:

```
easy_install -U pyglet
```

On Mac OS X and Linux you may need to run the above as a privileged user; for example:

```
sudo easy_install -U pyglet
```

Writing a pyglet application

Getting started with a new library or framework can be daunting, especially when presented with a large amount of reference material to read. This chapter gives a very quick introduction to pyglet without covering any of the details.

- *Hello, World*
- *Image viewer*
- *Handling mouse and keyboard events*
- *Playing sounds and music*
- *Where to next?*

Hello, World

We'll begin with the requisite "Hello, World" introduction. This program will open a window with some text in it and wait to be closed. You can find the entire program in the *examples/programming_guide/hello_world.py* file.

Begin by importing the *pyglet* package:

```
import pyglet
```

Create a *pyglet.window.Window* by calling its default constructor. The window will be visible as soon as it's created, and will have reasonable default values for all its parameters:

```
window = pyglet.window.Window()
```

To display the text, we'll create a *Label*. Keyword arguments are used to set the font, position and anchorage of the label:


```
label = pyglet.text.Label('Hello, world',
                           font_name='Times New Roman',
                           font_size=36,
                           x=window.width//2, y=window.height//2,
                           anchor_x='center', anchor_y='center')
```

An `on_draw()` event is dispatched to the window to give it a chance to redraw its contents. pyglet provides several ways to attach event handlers to objects; a simple way is to use a decorator:

```
@window.event
def on_draw():
    window.clear()
    label.draw()
```

Within the `on_draw()` handler the window is cleared to the default background color (black), and the label is drawn.

Finally, call:

```
pyglet.app.run()
```

To let pyglet respond to application events such as the mouse and keyboard. Your event handlers will now be called as required, and the `run()` method will return only when all application windows have been closed.

Note that earlier versions of pyglet required the application developer to write their own event-handling runloop. This is still possible, but discouraged; see *The application event loop* for details.

Image viewer

Most games will need to load and display images on the screen. In this example we'll load an image from the application's directory and display it within the window:

```
import pyglet

window = pyglet.window.Window()
image = pyglet.resource.image('kitten.jpg')

@window.event
def on_draw():
    window.clear()
    image.blit(0, 0)

pyglet.app.run()
```

We used the `image()` function to load the image, which automatically locates the file relative to the source file (rather than the working directory). To load an image not bundled with the application (for example, specified on the command line, you would use `pyglet.image.load()`).

The `blit()` method draws the image. The arguments `(0, 0)` tell pyglet to draw the image at pixel coordinates 0, 0 in the window (the lower-left corner).

The complete code for this example is located in *examples/programming_guide/image_viewer.py*.

Handling mouse and keyboard events

So far the only event used is the `on_draw()` event. To react to keyboard and mouse events, it's necessary to write and attach event handlers for these events as well:

```
import pyglet

window = pyglet.window.Window()

@window.event
def on_key_press(symbol, modifiers):
    print 'A key was pressed'

@window.event
def on_draw():
    window.clear()

pyglet.app.run()
```

Keyboard events have two parameters: the virtual key *symbol* that was pressed, and a bitwise combination of any *modifiers* that are present (for example, the CTRL and SHIFT keys).

The key symbols are defined in `pyglet.window.key`:

```
from pyglet.window import key

@window.event
def on_key_press(symbol, modifiers):
    if symbol == key.A:
        print 'The "A" key was pressed.'
    elif symbol == key.LEFT:
        print 'The left arrow key was pressed.'
    elif symbol == key.ENTER:
        print 'The enter key was pressed.'
```

See the `pyglet.window.key` documentation for a complete list of key symbols.

Mouse events are handled in a similar way:

```
from pyglet.window import mouse

@window.event
def on_mouse_press(x, y, button, modifiers):
    if button == mouse.LEFT:
        print 'The left mouse button was pressed.'
```

The `x` and `y` parameters give the position of the mouse when the button was pressed, relative to the lower-left corner of the window.

There are more than 20 event types that you can handle on a window. The easiest way to find the event name and parameters you need is to add the following line to your program:

```
window.push_handlers(pyglet.window.event.WindowEventLogger())
```

This will cause all events received on the window to be printed to the console.

An example program using keyboard and mouse events is in `examples/programming_guide/events.py`

Playing sounds and music

pyglet makes it easy to play and mix multiple sounds together in your game. The following example plays an MP3 file¹:

¹ MP3 and other compressed audio formats require AVbin to be installed (this is the default for the Windows and Mac OS X installers). Uncompressed WAV files can be played without AVbin.

```
import pyglet

music = pyglet.resource.media('music.mp3')
music.play()

pyglet.app.run()
```

As with the image loading example presented earlier, `media()` locates the sound file in the application's directory (not the working directory). If you know the actual filesystem path (either relative or absolute), use `load()`.

Short sounds, such as a gunfire shot used in a game, should be decoded in memory before they are used, so that they play more immediately and incur less of a CPU performance penalty. Specify `streaming=False` in this case:

```
sound = pyglet.resource.media('shot.wav', streaming=False)
sound.play()
```

The `examples/media_player.py` example demonstrates playback of streaming audio and video using pyglet. The `examples/noisy/noisy.py` example demonstrates playing many short audio samples simultaneously, as in a game.

Where to next?

The examples presented in this chapter should have given you enough information to get started writing simple arcade and point-and-click-based games.

The remainder of this programming guide goes into quite technical detail regarding some of pyglet's features. While getting started, it's recommended that you skim the beginning of each chapter but not attempt to read through the entire guide from start to finish.

To write 3D applications or achieve optimal performance in your 2D applications you'll need to work with OpenGL directly. The canonical references for OpenGL are [The OpenGL Programming Guide](#) and [The OpenGL Shading Language](#).

There are numerous examples of pyglet applications in the `examples/` directory of the documentation and source distributions. Keep checking <http://www.pyglet.org/> for more examples and tutorials as they are written.

Creating an OpenGL context

This section describes how to configure an OpenGL context. For most applications the information described here is far too low-level to be of any concern, however more advanced applications can take advantage of the complete control pyglet provides.

- *Displays, screens, configs and contexts*
 - *Contexts and configs*
 - *Displays*
 - *Screens*
- *OpenGL configuration options*
 - *The default configuration*
- *Simple context configuration*
- *Selecting the best configuration*
- *Sharing objects between contexts*

Displays, screens, configs and contexts

Fig. 1.1: Flow of construction, from the singleton Platform to a newly created Window with its Context.

Contexts and configs

When you draw on a window in pyglet, you are drawing to an OpenGL context. Every window has its own context, which is created when the window is created. You can access the window's context via its `context` attribute.

The context is created from an OpenGL configuration (or “config”), which describes various properties of the context such as what color format to use, how many buffers are available, and so on. You can access the config that was used to create a context via the context's `config` attribute.

For example, here we create a window using the default config and examine some of its properties:

```
>>> import pyglet
>>> window = pyglet.window.Window()
>>> context = window.context
>>> config = context.config
>>> config.double_buffer
c_int(1)
>>> config.stereo
c_int(0)
>>> config.sample_buffers
c_int(0)
```

Note that the values of the config's attributes are all ctypes instances. This is because the config was not specified by pyglet. Rather, it has been selected by pyglet from a list of configs supported by the system. You can make no guarantee that a given config is valid on a system unless it was provided to you by the system.

pyglet simplifies the process of selecting one of the system's configs by allowing you to create a “template” config which specifies only the values you are interested in. See *Simple context configuration* for details.

Displays

The system may actually support several different sets of configs, depending on which display device is being used. For example, a computer with two video cards would have not support the same configs on each card. Another example is using X11 remotely: the display device will support different configurations than the local driver. Even a single video card on the local computer may support different configs for the two monitors plugged in.

In pyglet, a `Display` is a collection of “screens” attached to a single display device. On Linux, the display device corresponds to the X11 display being used. On Windows and Mac OS X, there is only one display (as these operating systems present multiple video cards as a single virtual device).

There is a singleton class `Platform` which provides access to the display(s); this represents the computer on which your application is running. It is usually sufficient to use the default display:

```
>>> platform = pyglet.window.get_platform()
>>> display = platform.get_default_display()
```

On X11, you can specify the display string to use, for example to use a remotely connected display. The display string is in the same format as used by the `DISPLAY` environment variable:

```
>>> display = platform.get_display('remote:1.0')
```

You use the same string to specify a separate X11 screen ¹:

¹ Assuming Xinerama is not being used to combine the screens. If Xinerama is enabled, use screen 0 in the display string, and select a screen in the same manner as for Windows and Mac OS X.

```
>>> display = platform.get_display(':0.1')
```

Screens

Once you have obtained a display, you can enumerate the screens that are connected. A screen is the physical display medium connected to the display device; for example a computer monitor, TV or projector. Most computers will have a single screen, however dual-head workstations and laptops connected to a projector are common cases where more than one screen will be present.

In the following example the screens of a dual-head workstation are listed:

```
>>> for screen in display.get_screens():
...     print screen
...
XlibScreen(screen=0, x=1280, y=0, width=1280, height=1024, xinerama=1)
XlibScreen(screen=0, x=0, y=0, width=1280, height=1024, xinerama=1)
```

Because this workstation is running Linux, the returned screens are `XlibScreen`, a subclass of `Screen`. The `screen` and `xinerama` attributes are specific to Linux, but the `x`, `y`, `width` and `height` attributes are present on all screens, and describe the screen's geometry, as shown below.

Fig. 1.2: Example arrangement of screens and their reported geometry. Note that the primary display (marked “1”) is positioned on the right, according to this particular user's preference.

There is always a “default” screen, which is the first screen returned by `get_screens()`. Depending on the operating system, the default screen is usually the one that contains the taskbar (on Windows) or menu bar (on OS X). You can access this screen directly using `get_default_screen()`.

OpenGL configuration options

When configuring or selecting a `Config`, you do so based on the properties of that config. pyglet supports a fixed subset of the options provided by AGL, GLX, WGL and their extensions. In particular, these constraints are placed on all OpenGL configs:

- Buffers are always component (RGB or RGBA) color, never palette indexed.
- The “level” of a buffer is always 0 (this parameter is largely unsupported by modern OpenGL drivers anyway).
- There is no way to set the transparent color of a buffer (again, this GLX-specific option is not well supported).
- There is no support for pbuffers (equivalent functionality can be achieved much more simply and efficiently using framebuffer objects).

The visible portion of the buffer, sometimes called the color buffer, is configured with the following attributes:

buffer_size Number of bits per sample. Common values are 24 and 32, which each dedicate 8 bits per color component. A buffer size of 16 is also possible, which usually corresponds to 5, 6, and 5 bits of red, green and blue, respectively.

Usually there is no need to set this property, as the device driver will select a buffer size compatible with the current display mode by default.

red_size, blue_size, green_size, alpha_size These each give the number of bits dedicated to their respective color component. You should avoid setting any of the red, green or blue sizes, as these are determined by the driver based on the `buffer_size` property.

If you require an alpha channel in your color buffer (for example, if you are compositing in multiple passes) you should specify `alpha_size=8` to ensure that this channel is created.

sample_buffers and samples Configures the buffer for multisampling, in which more than one color sample is used to determine the color of each pixel, leading to a higher quality, antialiased image.

Enable multisampling by setting `sample_buffers=1`, then give the number of samples per pixel to use in `samples`. For example, `samples=2` is the fastest, lowest-quality multisample configuration. A higher-quality buffer (with a compromise in performance) is possible with `samples=4`.

Not all video hardware supports multisampling; you may need to make this a user-selectable option, or be prepared to automatically downgrade the configuration if the requested one is not available.

stereo Creates separate left and right buffers, for use with stereo hardware. Only specialised video hardware such as stereoscopic glasses will support this option. When used, you will need to manually render to each buffer, for example using `glDrawBuffers`.

double_buffer Create separate front and back buffers. Without double-buffering, drawing commands are immediately visible on the screen, and the user will notice a visible flicker as the image is redrawn in front of them.

It is recommended to set `double_buffer=True`, which creates a separate hidden buffer to which drawing is performed. When the `Window.flip` is called, the buffers are swapped, making the new drawing visible virtually instantaneously.

In addition to the color buffer, several other buffers can optionally be created based on the values of these properties:

depth_size A depth buffer is usually required for 3D rendering. The typical depth size is 24 bits. Specify 0 if you do not require a depth buffer.

stencil_size The stencil buffer is required for masking the other buffers and implementing certain volumetric shadowing algorithms. The typical stencil size is 8 bits; or specify 0 if you do not require it.

accum_red_size, accum_blue_size, accum_green_size, accum_alpha_size The accumulation buffer can be used for simple antialiasing, depth-of-field, motion blur and other compositing operations. Its use nowadays is being superseded by the use of floating-point textures, however it is still a practical solution for implementing these effects on older hardware.

If you require an accumulation buffer, specify 8 for each of these attributes (the alpha component is optional, of course).

aux_buffers Each auxilliary buffer is configured the same as the colour buffer. Up to four auxilliary buffers can typically be created. Specify 0 if you do not require any auxilliary buffers.

Like the accumulation buffer, auxilliary buffers are used less often nowadays as more efficient techniques such as render-to-texture are available. They are almost universally available on older hardware, though, where the newer techniques are not possible.

The default configuration

If you create a `Window` without specifying the context or config, pyglet will use a template config with the following properties:

Attribute	Value
<code>double_buffer</code>	<code>True</code>
<code>depth_size</code>	<code>24</code>

Simple context configuration

A context can only be created from a config that was provided by the system. Enumerating and comparing the attributes of all the possible configs is a complicated process, so pyglet provides a simpler interface based on “template” configs.

To get the config with the attributes you need, construct a `Config` and set only the attributes you are interested in. You can then supply this config to the `Window` constructor to create the context.

For example, to create a window with an alpha channel:

```
config = pyglet.gl.Config(alpha_size=8)
window = pyglet.window.Window(config=config)
```

It is sometimes necessary to create the context yourself, rather than letting the `Window` constructor do this for you. In this case use `get_best_config()` to obtain a “complete” config, which you can then use to create the context:

```
platform = pyglet.window.get_platform()
display = platform.get_default_display()
screen = display.get_default_screen()

template = pyglet.gl.Config(alpha_size=8)
config = screen.get_best_config(template)
context = config.create_context(None)
window = pyglet.window.Window(context=context)
```

Note that you cannot create a context directly from a template (any `Config` you constructed yourself). The `Window` constructor performs a similar process to the above to create the context if a template config is given.

Not all configs will be possible on all machines. The call to `get_best_config()` will raise `NoSuchConfigException` if the hardware does not support the requested attributes. It will never return a config that does not meet or exceed the attributes you specify in the template.

You can use this to support newer hardware features where available, but also accept a lesser config if necessary. For example, the following code creates a window with multisampling if possible, otherwise leaves multisampling off:

```
template = gl.Config(sample_buffers=1, samples=4)
try:
    config = screen.get_best_config(template)
except pyglet.window.NoSuchConfigException:
    template = gl.Config()
    config = screen.get_best_config(template)
window = pyglet.window.Window(config=config)
```

Selecting the best configuration

Allowing pyglet to select the best configuration based on a template is sufficient for most applications, however some complex programs may want to specify their own algorithm for selecting a set of OpenGL attributes.

You can enumerate a screen’s configs using the `get_matching_configs()` method. You must supply a template as a minimum specification, but you can supply an “empty” template (one with no attributes set) to get a list of all configurations supported by the screen.

In the following example, all configurations with either an auxiliary buffer or an accumulation buffer are printed:

```
platform = pyglet.window.get_platform()
display = platform.get_default_display()
screen = display.get_default_screen()

for config in screen.get_matching_configs(gl.Config()):
```

```
if config.aux_buffers or config.accum_red_size:
    print config
```

As well as supporting more complex configuration selection algorithms, enumeration allows you to efficiently find the maximum value of an attribute (for example, the maximum samples per pixel), or present a list of possible configurations to the user.

Sharing objects between contexts

Every window in pyglet has its own OpenGL context. Each context has its own OpenGL state, including the matrix stacks and current flags. However, contexts can optionally share their objects with one or more other contexts. Shareable objects include:

- Textures
- Display lists
- Shader programs
- Vertex and pixel buffer objects
- Framebuffer objects

There are two reasons for sharing objects. The first is to allow objects to be stored on the video card only once, even if used by more than one window. For example, you could have one window showing the actual game, with other “debug” windows showing the various objects as they are manipulated. Or, a set of widget textures required for a GUI could be shared between all the windows in an application.

The second reason is to avoid having to recreate the objects when a context needs to be recreated. For example, if the user wishes to turn on multisampling, it is necessary to recreate the context. Rather than destroy the old one and lose all the objects already created, you can

1. Create the new context, sharing object space with the old context, then
2. Destroy the old context. The new context retains all the old objects.

pyglet defines an `ObjectSpace`: a representation of a collection of objects used by one or more contexts. Each context has a single object space, accessible via its `object_space` attribute.

By default, all contexts share the same object space as long as at least one context using it is “alive”. If all the contexts sharing an object space are lost or destroyed, the object space will be destroyed also. This is why it is necessary to follow the steps outlined above for retaining objects when a context is recreated.

pyglet creates a hidden “shadow” context as soon as `pyglet.gl` is imported. By default, all windows will share object space with this shadow context, so the above steps are generally not needed. The shadow context also allows objects such as textures to be loaded before a window is created (see `shadow_window` in `pyglet.options` for further details).

When you create a `Context`, you tell pyglet which other context it will obtain an object space from. By default (when using the `Window` constructor to create the context) the most recently created context will be used. You can specify another context, or specify no context (to create a new object space) in the `Context` constructor.

It can be useful to keep track of which object space an object was created in. For example, when you load a font, pyglet caches the textures used and reuses them; but only if the font is being loaded on the same object space. The easiest way to do this is to set your own attributes on the `ObjectSpace` object.

In the following example, an attribute is set on the object space indicating that game objects have been loaded. This way, if the context is recreated, you can check for this attribute to determine if you need to load them again:


```
context = pyglet.gl.get_current_context()
object_space = context.object_space
object_space.my_game_objects_loaded = True
```

Avoid using attribute names on `ObjectSpace` that begin with "pyglet", they may conflict with an internal module.

The OpenGL interface

pyglet provides an interface to OpenGL and GLU. The interface is used by all of pyglet's higher-level API's, so that all rendering is done efficiently by the graphics card, rather than the operating system. You can access this interface directly; using it is much like using OpenGL from C.

The interface is a "thin-wrapper" around `libGL.so` on Linux, `opengl32.dll` on Windows and `OpenGL.framework` on OS X. The pyglet maintainers regenerate the interface from the latest specifications, so it is always up-to-date with the latest version and almost all extensions.

The interface is provided by the `pyglet.gl` package. To use it you will need a good knowledge of OpenGL, C and ctypes. You may prefer to use OpenGL without using ctypes, in which case you should investigate [PyOpenGL](#). [PyOpenGL](#) provides similar functionality with a more "Pythonic" interface, and will work with pyglet without any modification.

- *Using OpenGL*
- *Resizing the window*
- *Error checking*
- *Using extension functions*
- *Using multiple windows*
- *AGL, GLX and WGL*

Using OpenGL

Documentation of OpenGL and GLU are provided at the [OpenGL website](#) and (more comprehensively) in the [OpenGL Programming Guide](#).

Importing the package gives access to OpenGL, GLU, and all OpenGL registered extensions. This is sufficient for all but the most advanced uses of OpenGL:

```
from pyglet.gl import *
```

All function names and constants are identical to the C counterparts. For example, the following program draws a triangle on the screen:

```
from pyglet.gl import *

# Direct OpenGL commands to this window.
window = pyglet.window.Window()

@window.event
def on_draw():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glVertex2f(0, 0)
    glVertex2f(window.width, 0)
    glVertex2f(window.width, window.height)
```

```
glEnd()

pyglet.app.run()
```

Some OpenGL functions require an array of data. These arrays must be constructed as `ctypes` arrays of the correct type. The following example draw the same triangle as above, but uses a vertex array instead of the immediate-mode functions. Note the construction of the vertex array using a one-dimensional `ctypes` array of `GLfloat`:

```
from pyglet.gl import *

window = pyglet.window.Window()

vertices = [
    0, 0,
    window.width, 0,
    window.width, window.height]
vertices_gl = (GLfloat * len(vertices))(*vertices)

glEnableClientState(GL_VERTEX_ARRAY)
glVertexPointer(2, GL_FLOAT, 0, vertices_gl)

@window.event
def on_draw():
    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glDrawArrays(GL_TRIANGLES, 0, len(vertices) // 2)

pyglet.app.run()
```

Similar array constructions can be used to create data for vertex buffer objects, texture data, polygon stipple data and the map functions.

Resizing the window

pyglet sets up the viewport and an orthographic projection on each window automatically. It does this in a default `on_resize` handler defined on `Window`:

```
@window.event
def on_resize(width, height):
    glViewport(0, 0, width, height)
    glMatrixMode(gl.GL_PROJECTION)
    glLoadIdentity()
    glOrtho(0, width, 0, height, -1, 1)
    glMatrixMode(gl.GL_MODELVIEW)
```

If you need to define your own projection (for example, to use a 3-dimensional perspective projection), you should override this event with your own; for example:

```
@window.event
def on_resize(width, height):
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluPerspective(65, width / float(height), .1, 1000)
    glMatrixMode(GL_MODELVIEW)
    return pyglet.event.EVENT_HANDLED
```

Note that the `on_resize` handler is called for a window the first time it is displayed, as well as any time it is later resized.

Error checking

By default, pyglet calls `glGetError` after every GL function call (except where such a check would be invalid). If an error is reported, pyglet raises `GLEException` with the result of `gluErrorString` as the message.

This is very handy during development, as it catches common coding errors early on. However, it has a significant impact on performance, and is disabled when python is run with the `-O` option.

You can also disable this error check by setting the following option *before* importing `pyglet.gl` or `pyglet.window`:

```
# Disable error checking for increased performance
pyglet.options['debug_gl'] = False

from pyglet.gl import *
```

Setting the option after importing `pyglet.gl` will have no effect. Once disabled, there is no error-checking overhead in each GL call.

Using extension functions

Before using an extension function, you should check that the extension is implemented by the current driver. Typically this is done using `glGetString(GL_EXTENSIONS)`, but pyglet has a convenience module, `pyglet.gl.gl_info` that does this for you:

```
if pyglet.gl.gl_info.have_extension('GL_ARB_shadow'):
    # ... do shadow-related code.
else:
    # ... raise an exception, or use a fallback method
```

You can also easily check the version of OpenGL:

```
if pyglet.gl.gl_info.have_version(1,5):
    # We can assume all OpenGL 1.5 functions are implemented.
```

Remember to only call the `gl_info` functions after creating a window.

There is a corresponding `glu_info` module for checking the version and extensions of GLU.

nVidia often release hardware with extensions before having them registered officially. When you import `* from pyglet.gl` you import only the registered extensions. You can import the latest nVidia extensions with:

```
from pyglet.gl.glext_nv import *
```

Using multiple windows

pyglet allows you to create and display any number of windows simultaneously. Each will be created with its own OpenGL context, however all contexts will share the same texture objects, display lists, shader programs, and so on, by default¹. Each context has its own state and framebuffers.

There is always an active context (unless there are no windows). When using `pyglet.app.run` for the application event loop, pyglet ensures that the correct window is the active context before dispatching the `on_draw` or `on_resize` events.

In other cases, you can explicitly set the active context with `Window.switch_to`.

¹ Sometimes objects and lists cannot be shared between contexts; for example, when the contexts are provided by different video devices. This will usually only occur if you explicitly select different screens driven by different devices.

AGL, GLX and WGL

The OpenGL context itself is managed by an operating-system specific library: AGL on OS X, GLX under X11 and WGL on Windows. pyglet handles these details when a window is created, but you may need to use the functions directly (for example, to use pbuffers) or an extension function.

The modules are named `pyglet.gl.agl`, `pyglet.gl.glx` and `pyglet.gl.wgl`. You must only import the correct module for the running operating system:

```
if sys.platform.startswith('linux'):
    from pyglet.gl.glx import *
    glxCreatePbuffer(...)
elif sys.platform == 'darwin':
    from pyglet.gl.agl import *
    aglCreatePbuffer(...)
```

Alternatively you can use `pyglet.compat_platform` to support platforms that are compatible with platforms not officially supported by pyglet. For example FreeBSD systems will appear as `linux-compat` in `pyglet.compat_platform`.

There are convenience modules for querying the version and extensions of WGL and GLX named `pyglet.gl.wgl_info` and `pyglet.gl.glx_info`, respectively. AGL does not have such a module, just query the version of OS X instead.

If using GLX extensions, you can import `pyglet.gl.glxext_arb` for the registered extensions or `pyglet.gl.glxext_nv` for the latest nVidia extensions.

Similarly, if using WGL extensions, import `pyglet.gl.wglxext_arb` or `pyglet.gl.wglxext_nv`.

Graphics

At the lowest level, pyglet uses OpenGL to draw in windows. The OpenGL interface is exposed via the *pyglet.gl* module (see *The OpenGL interface*).

However, using the OpenGL interface directly for drawing graphics is difficult and inefficient. The *pyglet.graphics* module provides a simpler means for drawing graphics that uses vertex arrays and vertex buffer objects internally to deliver better performance.

- *Drawing primitives*
- *Vertex attributes*
- *Vertex lists*
 - *Updating vertex data*
 - *Data usage*
 - *Indexed vertex lists*
- *Batched rendering*
 - *Setting the OpenGL state*
 - *Hierarchical state*
 - *Sorting vertex lists*
- *Batches and groups in other modules*

Drawing primitives

The *pyglet.graphics* module draws the OpenGL primitive objects by a mode denoted by the constants

- `pyglet.gl.GL_POINTS`

- `pyglet.gl.GL_LINES`
- `pyglet.gl.GL_LINE_LOOP`
- `pyglet.gl.GL_LINE_STRIP`
- `pyglet.gl.GL_TRIANGLES`
- `pyglet.gl.GL_TRIANGLE_STRIP`
- `pyglet.gl.GL_TRIANGLE_FAN`
- `pyglet.gl.GL_QUADS`
- `pyglet.gl.GL_QUAD_STRIP`
- `pyglet.gl.GL_POLYGON`

See the [OpenGL Programming Guide](#) for a description of each of mode.

Each primitive is made up of one or more vertices. Each vertex is specified with either 2, 3 or 4 components (for 2D, 3D, or non-homogeneous coordinates). The data type of each component can be either int or float.

Use `pyglet.graphics.draw` to draw a primitive. The following example draws two points at coordinates (10, 15) and (30, 35):

```
pyglet.graphics.draw(2, pyglet.gl.GL_POINTS,
    ('v2i', (10, 15, 30, 35))
)
```

The first and second arguments to the function give the number of vertices to draw and the primitive mode, respectively. The third argument is a “data item”, and gives the actual vertex data.

However, because of the way the graphics API renders multiple primitives with shared state, `GL_POLYGON`, `GL_LINE_LOOP` and `GL_TRIANGLE_FAN` cannot be used — the results are undefined.

Alternatively, the `NV_primitive_restart` extension can be used if it is present. This also permits use of `GL_POLYGON`, `GL_LINE_LOOP` and `GL_TRIANGLE_FAN`. Unfortunately the extension is not provided by older video drivers, and requires indexed vertex lists.

Because vertex data can be supplied in several forms, a “format string” is required. In this case, the format string is “v2i”, meaning the vertex position data has two components (2D) and int type.

The following example has the same effect as the previous one, but uses floating point data and 3 components per vertex:

```
pyglet.graphics.draw(2, pyglet.gl.GL_POINTS,
    ('v3f', (10.0, 15.0, 0.0, 30.0, 35.0, 0.0))
)
```

Vertices can also be drawn out of order and more than once by using the `pyglet.graphics.draw_indexed` function. This requires a list of integers giving the indices into the vertex data. The following example draws the same two points as above, but indexes the vertices (sequentially):

```
pyglet.graphics.draw_indexed(2, pyglet.gl.GL_POINTS,
    [0, 1],
    ('v2i', (10, 15, 30, 35))
)
```

This second example is more typical; two adjacent triangles are drawn, and the shared vertices are reused with indexing:

```
pyglet.graphics.draw_indexed(4, pyglet.gl.GL_TRIANGLES,
    [0, 1, 2, 0, 2, 3],
    ('v2i', (100, 100,
              150, 100,
              150, 150,
              100, 150))
)
```

Note that the first argument gives the number of vertices in the data, not the number of indices (which is implicit on the length of the index list given in the third argument).

When using `GL_LINE_STRIP`, `GL_TRIANGLE_STRIP` or `GL_QUAD_STRIP` care must be taken to insert degenerate vertices at the beginning and end of each vertex list. For example, given the vertex list:

```
A, B, C, D
```

the correct vertex list to provide the vertex list is:

```
A, A, B, C, D, D
```

Vertex attributes

Besides the required vertex position, vertices can have several other numeric attributes. Each is specified in the format string with a letter, the number of components and the data type.

Each of the attributes is described in the table below with the set of valid format strings written as a regular expression (for example, `"v[234][if]"` means `"v2f"`, `"v3i"`, `"v4f"`, etc. are all valid formats).

Some attributes have a “recommended” format string, which is the most efficient form for the video driver as it requires less conversion.

Attribute	Formats	Recommended
Vertex position	<code>"v[234][sifd]"</code>	<code>"v[234]f"</code>
Color	<code>"c[34][bBsSiIfd]"</code>	<code>"c[34]B"</code>
Edge flag	<code>"e1[bB]"</code>	
Fog coordinate	<code>"f[1234][bBsSiIfd]"</code>	
Normal	<code>"n3[bsifd]"</code>	<code>"n3f"</code>
Secondary color	<code>"s[34][bBsSiIfd]"</code>	<code>"s[34]B"</code>
Texture coordinate	<code>"[0-31]?t[234][sifd]"</code>	<code>"[0-31]?t[234]f"</code>
Generic attribute	<code>"[0-15]g(n)?[1234][bBsSiIfd]"</code>	

The possible data types that can be specified in the format string are described below.

Format	Type	Python type
<code>"b"</code>	Signed byte	int
<code>"B"</code>	Unsigned byte	int
<code>"s"</code>	Signed short	int
<code>"S"</code>	Unsigned short	int
<code>"i"</code>	Signed int	int
<code>"I"</code>	Unsigned int	int
<code>"f"</code>	Single precision float	float
<code>"d"</code>	Double precision float	float

The following attributes are normalised to the range `[0, 1]`. The value is used as-is if the data type is floating-point. If the data type is byte, short or int, the value is divided by the maximum value representable by that type. For example, unsigned bytes are divided by 255 to get the normalised value.

- Color

- Secondary color
- Generic attributes with the "n" format given.

Texture coordinate attributes may optionally be preceded by a texture unit number. If unspecified, texture unit 0 (GL_TEXTURE0) is implied. It is the application's responsibility to ensure that the OpenGL version is adequate and that the specified texture unit is within the maximum allowed by the implementation.

Up to 16 generic attributes can be specified per vertex, and can be used by shader programs for any purpose (they are ignored in the fixed-function pipeline). For the other attributes, consult the OpenGL programming guide for details on their effects.

When using the `pyglet.graphics.draw` and related functions, attribute data is specified alongside the vertex position data. The following example reproduces the two points from the previous page, except that the first point is blue and the second green:

```
pyglet.graphics.draw(2, pyglet.gl.GL_POINTS,
    ('v2i', (10, 15, 30, 35)),
    ('c3B', (0, 0, 255, 0, 255, 0))
)
```

It is an error to provide more than one set of data for any attribute, or to mismatch the size of the initial data with the number of vertices specified in the first argument.

Vertex lists

There is a significant overhead in using `pyglet.graphics.draw` and `pyglet.graphics.draw_indexed` due to pyglet interpreting and formatting the vertex data for the video device. Usually the data drawn in each frame (of an animation) is identical or very similar to the previous frame, so this overhead is unnecessarily repeated.

A *VertexList* is a list of vertices and their attributes, stored in an efficient manner that's suitable for direct upload to the video card. On newer video cards (supporting OpenGL 1.5 or later) the data is actually stored in video memory.

Create a *VertexList* for a set of attributes and initial data with `pyglet.graphics.vertex_list`. The following example creates a vertex list with the two coloured points used in the previous page:

```
vertex_list = pyglet.graphics.vertex_list(2,
    ('v2i', (10, 15, 30, 35)),
    ('c3B', (0, 0, 255, 0, 255, 0))
)
```

To draw the vertex list, call its *VertexList.draw* method:

```
vertex_list.draw(pyglet.gl.GL_POINTS)
```

Note that the primitive mode is given to the draw method, not the vertex list constructor. Otherwise the *vertex_list* method takes the same arguments as `pyglet.graphics.draw`, including any number of vertex attributes.

Because vertex lists can reside in video memory, it is necessary to call the *delete* method to release video resources if the vertex list isn't going to be used any more (there's no need to do this if you're just exiting the process).

Updating vertex data

The data in a vertex list can be modified. Each vertex attribute (including the vertex position) appears as an attribute on the *VertexList* object. The attribute names are given in the following table.

Vertex attribute	Object attribute
Vertex position	<code>vertices</code>
Color	<code>colors</code>
Edge flag	<code>edge_flags</code>
Fog coordinate	<code>fog_coords</code>
Normal	<code>normals</code>
Secondary color	<code>secondary_colors</code>
Texture coordinate	<code>tex_coords</code> ¹
Generic attribute	<i>Inaccessible</i>

In the following example, the vertex positions of the vertex list are updated by replacing the `vertices` attribute:

```
vertex_list.vertices = [20, 25, 40, 45]
```

The attributes can also be selectively updated in-place:

```
vertex_list.vertices[:2] = [30, 35]
```

Similarly, the color attribute of the vertex can be updated:

```
vertex_list.colors[:3] = [255, 0, 0]
```

For large vertex lists, updating only the modified vertices can have a performance benefit, especially on newer graphics cards.

Attempting to set the attribute list to a different size will cause an error (not necessarily immediately, either). To resize the vertex list, call `VertexList.resize` with the new vertex count. Be sure to fill in any newly uninitialised data after resizing the vertex list.

Since vertex lists are mutable, you may not necessarily want to initialise them with any particular data. You can specify just the format string in place of the `(format, data)` tuple in the data arguments `vertex_list` function. The following example creates a vertex list of 1024 vertices with positional, color, texture coordinate and normal attributes:

```
vertex_list = pyglet.graphics.vertex_list(1024, 'v3f', 'c4B', 't2f', 'n3f')
```

Data usage

By default, pyglet assumes vertex data will be updated less often than it is drawn, but more often than just during initialisation. You can override this assumption for each attribute by affixing a usage specification onto the end of the format string, detailed in the following table:

Usage	Description
<code>"/static"</code>	Data is never or rarely modified after initialisation
<code>"/dynamic"</code>	Data is occasionally modified (default)
<code>"/stream"</code>	Data is updated every frame

In the following example a vertex list is created in which the positional data is expected to change every frame, but the color data is expected to remain relatively constant:

```
vertex_list = pyglet.graphics.vertex_list(1024, 'v3f/stream', 'c4B/static')
```

The usage specification affects how pyglet lays out vertex data in memory, whether or not it's stored on the video card, and is used as a hint to OpenGL. Specifying a usage does not affect what operations are possible with a vertex list (a `static` attribute can still be modified), and may only have performance benefits on some hardware.

¹Only texture coordinates for texture unit 0 are accessible through this attribute.

Indexed vertex lists

IndexedVertexList performs the same role as *VertexList*, but for indexed vertices. Use *pyglet.graphics.vertex_list_indexed* to construct an indexed vertex list, and update the *IndexedVertexList.indices* sequence to change the indices.

Batched rendering

For optimal OpenGL performance, you should render as many vertex lists as possible in a single *draw* call. Internally, pyglet uses *VertexDomain* and *IndexedVertexDomain* to keep vertex lists that share the same attribute formats in adjacent areas of memory. The entire domain of vertex lists can then be drawn at once, without calling *VertexList.draw* on each individual list.

It is quite difficult and tedious to write an application that manages vertex domains itself, though. In addition to maintaining a vertex domain for each set of attribute formats, domains must also be separated by primitive mode and required OpenGL state.

The *Batch* class implements this functionality, grouping related vertex lists together and sorting by OpenGL state automatically. A batch is created with no arguments:

```
batch = pyglet.graphics.Batch()
```

Vertex lists can now be created with the *Batch.add* and *Batch.add_indexed* methods instead of *pyglet.graphics.vertex_list* and *pyglet.graphics.vertex_list_indexed* functions. Unlike the module functions, these methods accept a *mode* parameter (the primitive mode) and a *group* parameter (described below).

The two coloured points from previous pages can be added to a batch as a single vertex list with:

```
vertex_list = batch.add(2, pyglet.gl.GL_POINTS, None,
    ('v2i', (10, 15, 30, 35)),
    ('c3B', (0, 0, 255, 0, 255, 0))
)
```

The resulting *vertex_list* can be modified as described in the previous section. However, instead of calling *VertexList.draw* to draw it, call *Batch.draw* to draw all vertex lists contained in the batch at once:

```
batch.draw()
```

For batches containing many vertex lists this gives a significant performance improvement over drawing individual vertex lists.

To remove a vertex list from a batch, call *VertexList.delete*.

Setting the OpenGL state

In order to achieve many effects in OpenGL one or more global state parameters must be set. For example, to enable and bind a texture requires:

```
from pyglet.gl import *
glEnable(texture.target)
glBindTexture(texture.target, texture.id)
```

before drawing vertex lists, and then:

```
glDisable(texture.target)
```

afterwards to avoid interfering with later drawing commands.

With a *Group* these state changes can be encapsulated and associated with the vertex lists they affect. Subclass *Group* and override the *Group.set_state* and *Group.unset_state* methods to perform the required state changes:

```
class CustomGroup(pyglet.graphics.Group):
    def set_state(self):
        glEnable(texture.target)
        glBindTexture(texture.target, texture.id)

    def unset_state(self):
        glDisable(texture.target)
```

An instance of this group can now be attached to vertex lists in the batch:

```
custom_group = CustomGroup()
vertex_list = batch.add(2, pyglet.gl.GL_POINTS, custom_group,
    ('v2i', (10, 15, 30, 35)),
    ('c3B', (0, 0, 255, 0, 255, 0))
)
```

The *Batch* ensures that the appropriate *set_state* and *unset_state* methods are called before and after the vertex lists that use them.

Hierarchical state

Groups have a *parent* attribute that allows them to be implicitly organised in a tree structure. If groups **B** and **C** have parent **A**, then the order of *set_state* and *unset_state* calls for vertex lists in a batch will be:

```
A.set_state()
# Draw A vertices
B.set_state()
# Draw B vertices
B.unset_state()
C.set_state()
# Draw C vertices
C.unset_state()
A.unset_state()
```

This is useful to group state changes into as few calls as possible. For example, if you have a number of vertex lists that all need texturing enabled, but have different bound textures, you could enable and disable texturing in the parent group and bind each texture in the child groups. The following example demonstrates this:

```
class TextureEnableGroup(pyglet.graphics.Group):
    def set_state(self):
        glEnable(GL_TEXTURE_2D)

    def unset_state(self):
        glDisable(GL_TEXTURE_2D)

texture_enable_group = TextureEnableGroup()

class TextureBindGroup(pyglet.graphics.Group):
    def __init__(self, texture):
        super(TextureBindGroup, self).__init__(parent=texture_enable_group)
        assert texture.target == GL_TEXTURE_2D
        self.texture = texture
```

```

def set_state(self):
    glBindTexture(GL_TEXTURE_2D, self.texture.id)

# No unset_state method required.

def __eq__(self, other):
    return (self.__class__ is other.__class__ and
            self.texture.id == other.texture.id and
            self.texture.target == other.texture.target and
            self.parent == other.parent)

def __hash__(self):
    return hash((self.texture.id, self.texture.target))

batch.add(4, GL_QUADS, TextureBindGroup(texture1), 'v2f', 't2f')
batch.add(4, GL_QUADS, TextureBindGroup(texture2), 'v2f', 't2f')
batch.add(4, GL_QUADS, TextureBindGroup(texture1), 'v2f', 't2f')

```

Note the use of an `__eq__` method on the group to allow *Batch* to merge the two *TextureBindGroup* identical instances.

Sorting vertex lists

VertexDomain does not attempt to keep vertex lists in any particular order. So, any vertex lists sharing the same primitive mode, attribute formats and group will be drawn in an arbitrary order. However, *Batch* will sort *Group* objects sharing the same parent by their `__cmp__` method. This allows groups to be ordered.

The *OrderedGroup* class is a convenience group that does not set any OpenGL state, but is parameterised by an integer giving its draw order. In the following example a number of vertex lists are grouped into a “background” group that is drawn before the vertex lists in the “foreground” group:

```

background = pyglet.graphics.OrderedGroup(0)
foreground = pyglet.graphics.OrderedGroup(1)

batch.add(4, GL_QUADS, foreground, 'v2f')
batch.add(4, GL_QUADS, background, 'v2f')
batch.add(4, GL_QUADS, foreground, 'v2f')
batch.add(4, GL_QUADS, background, 'v2f', 'c4B')

```

By combining hierarchical groups with ordered groups it is possible to describe an entire scene within a single *Batch*, which then renders it as efficiently as possible.

Batches and groups in other modules

The *Sprite*, *Label* and *TextLayout* classes all accept `batch` and `group` parameters in their constructors. This allows you to add any of these higher-level pyglet drawables into arbitrary places in your rendering code.

For example, multiple sprites can be grouped into a single batch and then drawn at once, instead of calling *Sprite.draw* on each one individually:

```

batch = pyglet.graphics.Batch()
sprites = [pyglet.sprite.Sprite(image, batch=batch) for i in range(100)]

batch.draw()

```

The `group` parameter can be used to set the drawing order (and hence which objects overlap others) within a single batch, as described on the previous page.

In general you should batch all drawing objects into as few batches as possible, and use groups to manage the draw order and other OpenGL state changes for optimal performance. If you are creating your own drawable classes, consider adding `batch` and `group` parameters in a similar way.

Windowing

A *Window* in pyglet corresponds to a top-level window provided by the operating system. Windows can be floating (overlapped with other application windows) or fullscreen.

- *Creating a window*
 - *Context configuration*
 - *Fullscreen windows*
- *Size and position*
- *Appearance*
 - *Window style*
 - *Caption*
 - *Icon*
- *Visibility*
- *Subclassing Window*
- *Windows and OpenGL contexts*
 - *Double-buffering*
 - *Vertical retrace synchronisation*

Creating a window

If the *Window* constructor is called with no arguments, defaults will be assumed for all parameters:

```
window = pyglet.window.Window()
```

The default parameters used are:

- The window will have a size of 640x480, and not be resizable.
- A default context will be created using template config described in *OpenGL configuration options*.
- The window caption will be the name of the executing Python script (i.e., `sys.argv[0]`).

Windows are visible as soon as they are created, unless you give the `visible=False` argument to the constructor. The following example shows how to create and display a window in two steps:

```
window = pyglet.window.Window(visible=False)
# ... perform some additional initialisation
window.set_visible()
```

Context configuration

The context of a window cannot be changed once created. There are several ways to control the context that is created:

- Supply an already-created *Context* using the `context` argument:

```
context = config.create_context(share)
window = pyglet.window.Window(context=context)
```

- Supply a complete *Config* obtained from a *Screen* using the `config` argument. The context will be created from this config and will share object space with the most recently created existing context:

```
config = screen.get_best_config(template)
window = pyglet.window.Window(config=config)
```

- Supply a template *Config* using the `config` argument. The context will use the best config obtained from the default screen of the default display:

```
config = gl.Config(double_buffer=True)
window = pyglet.window.Window(config=config)
```

- Specify a *Screen* using the `screen` argument. The context will use a config created from default template configuration and this screen:

```
screen = display.get_screens()[screen_number]
window = pyglet.window.Window(screen=screen)
```

- Specify a *Display* using the `display` argument. The default screen on this display will be used to obtain a context using the default template configuration:

```
display = platform.get_display(display_name)
window = pyglet.window.Window(display=display)
```

If a template *Config* is given, a *Screen* or *Display* may also be specified; however any other combination of parameters overconstrains the configuration and some parameters will be ignored.

Fullscreen windows

If the `fullscreen=True` argument is given to the window constructor, the window will draw to an entire screen rather than a floating window. No window border or controls will be shown, so you must ensure you provide some other means to exit the application.

By default, the default screen on the default display will be used, however you can optionally specify another screen to use instead. For example, the following code creates a fullscreen window on the secondary screen:

```
screens = display.get_screens()
window = pyglet.window.Window(fullscreen=True, screens[1])
```

There is no way to create a fullscreen window that spans more than one window (for example, if you wanted to create an immersive 3D environment across multiple monitors). Instead, you should create a separate fullscreen window for each screen and attach identical event handlers to all windows.

Windows can be toggled in and out of fullscreen mode with the `set_fullscreen` method. For example, to return to windowed mode from fullscreen:

```
window.set_fullscreen(False)
```

The previous window size and location, if any, will attempt to be restored, however the operating system does not always permit this, and the window may have relocated.

Size and position

This section applies only to windows that are not fullscreen. Fullscreen windows always have the width and height of the screen they fill.

You can specify the size of a window as the first two arguments to the window constructor. In the following example, a window is created with a width of 800 pixels and a height of 600 pixels:

```
window = pyglet.window.Window(800, 600)
```

The “size” of a window refers to the drawable space within it, excluding any additional borders or title bar drawn by the operating system.

You can allow the user to resize your window by specifying `resizable=True` in the constructor. If you do this, you may also want to handle the *on_resize* event:

```
window = pyglet.window.Window(resizable=True)

@window.event
def on_resize(width, height):
    print 'The window was resized to %dx%d' % (width, height)
```

You can specify a minimum and maximum size that the window can be resized to by the user with the *set_minimum_size* and *set_maximum_size* methods:

```
window.set_minimum_size(320, 200)
window.set_maximum_size(1024, 768)
```

The window can also be resized programatically (even if the window is not user-resizable) with the *set_size* method:

```
window.set_size(800, 600)
```

The window will initially be positioned by the operating system. Typically, it will use its own algorithm to locate the window in a place that does not block other application windows, or cascades with them. You can manually adjust the position of the window using the *get_position* and *set_position* methods:

```
x, y = window.get_location()
window.set_location(x + 20, y + 20)
```

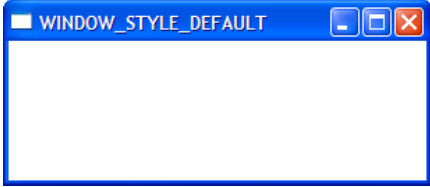
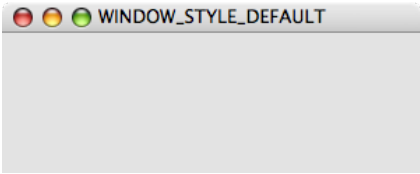
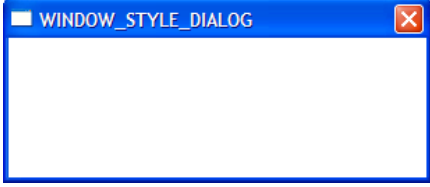
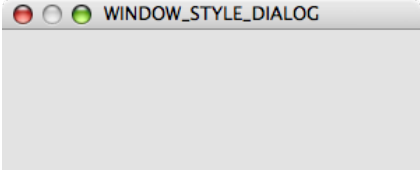
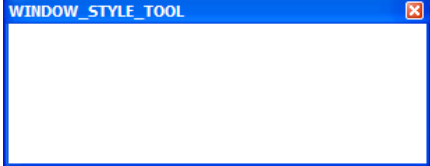
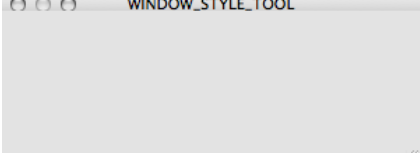
Note that unlike the usual coordinate system in pyglet, the window location is relative to the top-left corner of the desktop, as shown in the following diagram:

Fig. 1.3: The position and size of the window relative to the desktop.

Appearance

Window style

Non-fullscreen windows can be created in one of four styles: default, dialog, tool or borderless. Examples of the appearances of each of these styles under Windows XP and Mac OS X 10.4 are shown below.

Style	Windows XP	Mac OS X
<i>WINDOW_STYLE_DEFAULT</i>		
<i>WINDOW_STYLE_DIALOG</i>		
<i>WINDOW_STYLE_TOOL</i>		

Non-resizable variants of these window styles may appear slightly different (for example, the maximize button will either be disabled or absent).

Besides the change in appearance, the window styles affect how the window behaves. For example, tool windows do not usually appear in the task bar and cannot receive keyboard focus. Dialog windows cannot be minimized. Selecting the appropriate window style for your windows means your application will behave correctly for the platform on which it is running, however that behaviour may not be consistent across Windows, Linux and Mac OS X.

The appearance and behaviour of windows in Linux will vary greatly depending on the distribution, window manager and user preferences.

Borderless windows (*WINDOW_STYLE_BORDERLESS*) are not decorated by the operating system at all, and have no way to be resized or moved around the desktop. These are useful for implementing splash screens or custom window borders.

You can specify the style of the window in the *Window* constructor. Once created, the window style cannot be altered:

```
window = pyglet.window.Window(style=window.Window.WINDOW_STYLE_DIALOG)
```

Caption

The window's caption appears in its title bar and task bar icon (on Windows and some Linux window managers). You can set the caption during window creation or at any later time using the *set_caption* method:

```
window = pyglet.window.Window(caption='Initial caption')
window.set_caption('A different caption')
```

Icon

The window icon appears in the title bar and task bar icon on Windows and Linux, and in the dock icon on Mac OS X. Dialog and tool windows do not necessarily show their icon.

Windows, Mac OS X and the Linux window managers each have their own preferred icon sizes:

Windows XP

- A 16x16 icon for the title bar and task bar.
- A 32x32 icon for the Alt+Tab switcher.

Mac OS X

- Any number of icons of resolutions 16x16, 24x24, 32x32, 48x48, 72x72 and 128x128. The actual image displayed will be interpolated to the correct size from those provided.

Linux

- No constraints, however most window managers will use a 16x16 and a 32x32 icon in the same way as Windows XP.

The `Window.set_icon` method allows you to set any number of images as the icon. pyglet will select the most appropriate ones to use and apply them to the window. If an alternate size is required but not provided, pyglet will scale the image to the correct size using a simple interpolation algorithm.

The following example provides both a 16x16 and a 32x32 image as the window icon:

```
window = pyglet.window.Window()
icon1 = pyglet.image.load('16x16.png')
icon2 = pyglet.image.load('32x32.png')
window.set_icon(icon1, icon2)
```

You can use images in any format supported by pyglet, however it is recommended to use a format that supports alpha transparency such as PNG. Windows .ico files are supported only on Windows, so their use is discouraged. Mac OS X .icons files are not supported at all.

Note that the icon that you set at runtime need not have anything to do with the application icon, which must be encoded specially in the application binary (see *Self-contained executables*).

Visibility

Windows have several states of visibility. Already shown is the *visible* property which shows or hides the window.

Windows can be minimized, which is equivalent to hiding them except that they still appear on the taskbar (or are minimised to the dock, on OS X). The user can minimize a window by clicking the appropriate button in the title bar. You can also programmatically minimize a window using the *minimize* method (there is also a corresponding *maximize* method).

When a window is made visible the *on_show* event is triggered. When it is hidden the *on_hide* event is triggered. On Windows and Linux these events will only occur when you manually change the visibility of the window or when the window is minimized or restored. On Mac OS X the user can also hide or show the window (affecting visibility) using the Command+H shortcut.

Subclassing Window

A useful pattern in pyglet is to subclass *Window* for each type of window you will display, or as your main application class. There are several benefits:

- You can load font and other resources from the constructor, ensuring the OpenGL context has already been created.
- You can add event handlers simply by defining them on the class. The *on_resize* event will be called as soon as the window is created (this doesn't usually happen, as you must create the window before you can attach event handlers).

- There is reduced need for global variables, as you can maintain application state on the window.

The following example shows the same “Hello World” application as presented in *Writing a pyglet application*, using a subclass of *Window*:

```
class HelloWorldWindow(pyglet.window.Window):
    def __init__(self):
        super(HelloWorldWindow, self).__init__()

        self.label = pyglet.text.Label('Hello, world!')

    def on_draw(self):
        self.clear()
        self.label.draw()

if __name__ == '__main__':
    window = HelloWorldWindow()
    pyglet.app.run()
```

This example program is located in *examples/programming_guide/window_subclass.py*.

Windows and OpenGL contexts

Every window in pyglet has an associated OpenGL context. Specifying the configuration of this context has already been covered in *Creating a window*. Drawing into the OpenGL context is the only way to draw into the window’s client area.

Double-buffering

If the window is double-buffered (i.e., the configuration specified `double_buffer=True`, the default), OpenGL commands are applied to a hidden back buffer. This back buffer can be copied to the window using the *flip* method. If you are using the standard *pyglet.app.run* or *pyglet.app.EventLoop* event loop, this is taken care of automatically after each *on_draw* event.

If the window is not double-buffered, the *flip* operation is unnecessary, and you should remember only to call *glFlush* to ensure buffered commands are executed.

Vertical retrace synchronisation

Double-buffering eliminates one cause of flickering: the user is unable to see the image as it painted, only the final rendering. However, it does introduce another source of flicker known as “tearing”.

Tearing becomes apparent when displaying fast-moving objects in an animation. The buffer flip occurs while the video display is still reading data from the framebuffer, causing the top half of the display to show the previous frame while the bottom half shows the updated frame. If you are updating the framebuffer particularly quickly you may notice three or more such “tears” in the display.

pyglet provides a way to avoid tearing by synchronising buffer flips to the video refresh rate. This is enabled by default, but can be set or unset manually at any time with the *vsync* (vertical retrace synchronisation) property. A window is created with *vsync* initially disabled in the following example:

```
window = pyglet.window.Window(vsync=False)
```

It is usually desirable to leave *vsync* enabled, as it results in flicker-free animation. There are some use-cases where you may want to disable it, for example:

- Profiling an application. Measuring the time taken to perform an operation will be affected by the time spent waiting for the video device to refresh, which can throw off results. You should disable vsync if you are measuring the performance of your application.
- If you cannot afford for your application to block. If your application run loop needs to quickly poll a hardware device, for example, you may want to avoid blocking with vsync.

Note that some older video cards do not support the required extensions to implement vsync; this will appear as a warning on the console but is otherwise ignored.

The application event loop

In order to let pyglet process operating system events such as mouse and keyboard events, applications need to enter an application event loop. The event loop continuously checks for new events, dispatches those events, and updates the contents of all open windows.

pyglet provides an application event loop that is tuned for performance and low power usage on Windows, Linux and Mac OS X. Most applications need only call:

```
pyglet.app.run()
```

to enter the event loop after creating their initial set of windows and attaching event handlers. The `run` function does not return until all open windows have been closed, or until `pyglet.app.exit()` is called.

The pyglet application event loop dispatches window events (such as for mouse and keyboard input) as they occur and dispatches the `on_draw` event to each window after every iteration through the loop.

To have additional code run periodically or every iteration through the loop, schedule functions on the clock (see *Scheduling functions for future execution*). pyglet ensures that the loop iterates only as often as necessary to fulfil all scheduled functions and user input.

Customising the event loop

The pyglet event loop is encapsulated in the `EventLoop` class, which provides several hooks that can be overridden for customising its behaviour. This is recommended only for advanced users – typical applications and games are unlikely to require this functionality.

To use the `EventLoop` class directly, instantiate it and call `run`:

```
pyglet.app.EventLoop().run()
```

Only one `EventLoop` can be running at a time; when the `run` method is called the module variable `pyglet.app.event_loop` is set to the running instance. Other pyglet modules such as `pyglet.window` depend on this.

Event loop events

You can listen for several events on the event loop instance. The most useful of these is `on_window_close`, which is dispatched whenever a window is closed. The default handler for this event exits the event loop if there are no more windows. The following example overrides this behaviour to exit the application whenever any window is closed:

```
event_loop = pyglet.app.EventLoop()

@event_loop.event
def on_window_close(window):
    event_loop.exit()
    return pyglet.event.EVENT_HANDLED
```

```
event_loop.run()
```

Overriding the default idle policy

The *EventLoop.idle* method is called every iteration of the event loop. It is responsible for calling scheduled clock functions, redrawing windows, and deciding how idle the application is. You can override this method if you have specific requirements for tuning the performance of your application; especially if it uses many windows.

The default implementation has the following algorithm:

1. Call *clock.tick* with `poll=True` to call any scheduled functions.
2. Dispatch the *on_draw* event and call *flip* on every open window.
3. Return the value of *clock.get_sleep_time*.

The return value of the method is the number of seconds until the event loop needs to iterate again (unless there is an earlier user-input event); or `None` if the loop can wait for input indefinitely.

Note that this default policy causes every window to be redrawn during every user event – if you have more knowledge about which events have an effect on which windows you can improve on the performance of this method.

Dispatching events manually

Earlier versions of pyglet and certain other windowing toolkits such as PyGame and SDL require the application developer to write their own event loop. This “manual” event loop is usually just an inconvenience compared to *pyglet.app.run*, but can be necessary in some situations when combining pyglet with other toolkits.

A simple event loop usually has the following form:

```
while True:
    pyglet.clock.tick()

    for window in pyglet.app.windows:
        window.switch_to()
        window.dispatch_events()
        window.dispatch_event('on_draw')
        window.flip()
```

The *dispatch_events* method checks the window’s operating system event queue for user input and dispatches any events found. The method does not wait for input – if there are no events pending, control is returned to the program immediately.

The call to *pyglet.clock.tick()* is required for ensuring scheduled functions are called, including the internal data pump functions for playing sounds and video.

Developers are strongly discouraged from writing pyglet applications with event loops like this:

- The *EventLoop* class provides plenty of hooks for most toolkits to be integrated without needing to resort to a manual event loop.
- Because *EventLoop* is tuned for specific operating systems, it is more responsive to user events, and continues calling clock functions while windows are being resized, and (on Mac OS X) the menu bar is being tracked.
- It is difficult to write a manual event loop that does not consume 100% CPU while still remaining responsive to user input.

The capability for writing manual event loops remains for legacy support and extreme circumstances.

The pyglet event framework

The *pyglet.window*, *pyglet.media*, *pyglet.app* and *pyglet.text* modules make use of a consistent event pattern, which provides several ways to attach event handlers to objects. You can also reuse this pattern in your own classes easily.

Throughout this documentation, an “event dispatcher” is an object that has events it needs to notify other objects about, and an “event handler” is some code that can be attached to a dispatcher.

- *Setting event handlers*
- *Stacking event handlers*
- *Creating your own event dispatcher*
 - *Implementing the Observer pattern*
 - *Documenting events*

Setting event handlers

An event handler is simply a function with a formal parameter list corresponding to the event type. For example, the *Window.on_resize* event has the parameters (*width*, *height*), so an event handler for this event could be:

```
def on_resize(width, height):  
    pass
```

The *Window* class subclasses *EventDispatcher*, which enables it to have event handlers attached to it. The simplest way to attach an event handler is to set the corresponding attribute on the object:

```
window = pyglet.window.Window()  
  
def on_resize(width, height):  
    pass  
window.on_resize = on_resize
```

While this technique is straight-forward, it requires you to write the name of the event three times for the one function, which can get tiresome. pyglet provides a shortcut using the *event* decorator:

```
window = window.Window()  
  
@window.event  
def on_resize(width, height):  
    pass
```

This is not entirely equivalent to setting the event handler directly on the object. If the object already had an event handler, using *@event* will add the handler to the object, rather than replacing it. The next section describes this functionality in detail.

As shown in *Subclassing Window*, you can also attach event handlers by subclassing the event dispatcher and adding the event handler as a method:

```
class MyWindow(pyglet.window.Window):  
    def on_resize(self, width, height):  
        pass
```

Stacking event handlers

It is often convenient to attach more than one event handler for an event. *EventDispatcher* allows you to stack event handlers upon one another, rather than replacing them outright. The event will propagate from the top of the stack to

the bottom, but can be stopped by any handler along the way.

To push an event handler onto the stack, use the *push_handlers* method:

```
def on_key_press(symbol, modifiers):
    if symbol == key.SPACE:
        fire_laser()

window.push_handlers(on_key_press)
```

As a convenience, the `@event` decorator can be used as an alternative to *push_handlers*:

```
@window.event
def on_key_press(symbol, modifiers):
    if symbol == key.SPACE:
        fire_laser()
```

One use for pushing handlers instead of setting them is to handle different parameterisations of events in different functions. In the above example, if the spacebar is pressed, the laser will be fired. After the event handler returns control is passed to the next handler on the stack, which on a *Window* is a function that checks for the ESC key and sets the `has_exit` attribute if it is pressed. By pushing the event handler instead of setting it, the application keeps the default behaviour while adding additional functionality.

You can prevent the remaining event handlers in the stack from receiving the event by returning a true value. The following event handler, when pushed onto the window, will prevent the escape key from exiting the program:

```
def on_key_press(symbol, modifiers):
    if symbol == key.ESCAPE:
        return True

window.push_handlers(on_key_press)
```

You can push more than one event handler at a time, which is especially useful when coupled with the *pop_handlers* function. In the following example, when the game starts some additional event handlers are pushed onto the stack. When the game ends (perhaps returning to some menu screen) the handlers are popped off in one go:

```
def start_game():
    def on_key_press(symbol, modifiers):
        print 'Key pressed in game'
        return True

    def on_mouse_press(x, y, button, modifiers):
        print 'Mouse button pressed in game'
        return True

    window.push_handlers(on_key_press, on_mouse_press)

def end_game():
    window.pop_handlers()
```

Note that you do not specify which handlers to pop off the stack – the entire top “level” (consisting of all handlers specified in a single call to *push_handlers*) is popped.

You can apply the same pattern in an object-oriented fashion by grouping related event handlers in a single class. In the following example, a `GameEventHandler` class is defined. An instance of that class can be pushed on and popped off of a window:

```
class GameEventHandler(object):
    def on_key_press(self, symbol, modifiers):
        print 'Key pressed in game'
```

```
        return True

    def on_mouse_press(self, x, y, button, modifiers):
        print 'Mouse button pressed in game'
        return True

game_handlers = GameEventHandler()

def start_game():
    window.push_handlers(game_handlers)

def stop_game():
    window.pop_handlers()
```

Creating your own event dispatcher

pyglet provides only the *Window* and *Player* event dispatchers, but exposes a public interface for creating and dispatching your own events.

The steps for creating an event dispatcher are:

1. Subclass *EventDispatcher*
2. Call the *register_event_type* class method on your subclass for each event your subclass will recognise.
3. Call *dispatch_event* to create and dispatch an event as needed.

In the following example, a hypothetical GUI widget provides several events:

```
class ClankingWidget(pyglet.event.EventDispatcher):
    def clank(self):
        self.dispatch_event('on_clank')

    def click(self, clicks):
        self.dispatch_event('on_clicked', clicks)

    def on_clank(self):
        print 'Default clank handler.'

ClankingWidget.register_event_type('on_clank')
ClankingWidget.register_event_type('on_clicked')
```

Event handlers can then be attached as described in the preceding sections:

```
widget = ClankingWidget()

@widget.event
def on_clank():
    pass

@widget.event
def on_clicked(clicks):
    pass

def override_on_clicked(clicks):
    pass

widget.push_handlers(on_clicked=override_on_clicked)
```

The *EventDispatcher* takes care of propagating the event to all attached handlers or ignoring it if there are no handlers for that event.

There is zero instance overhead on objects that have no event handlers attached (the event stack is created only when required). This makes *EventDispatcher* suitable for use even on light-weight objects that may not always have handlers. For example, *Player* is an *EventDispatcher* even though potentially hundreds of these objects may be created and destroyed each second, and most will not need an event handler.

Implementing the Observer pattern

The Observer design pattern, also known as Publisher/Subscriber, is a simple way to decouple software components. It is used extensively in many large software projects; for example, Java’s AWT and Swing GUI toolkits and the Python logging module; and is fundamental to any Model-View-Controller architecture.

EventDispatcher can be used to easily add observable components to your application. The following example recreates the *ClockTimer* example from *Design Patterns* (pages 300-301), though without needing the bulky *Attach*, *Detach* and *Notify* methods:

```
# The subject
class ClockTimer(pyglet.event.EventDispatcher):
    def tick(self):
        self.dispatch_event('on_update')
ClockTimer.register_event_type('on_update')

# Abstract observer class
class Observer(object):
    def __init__(self, subject):
        subject.push_handlers(self)

# Concrete observer
class DigitalClock(Observer):
    def on_update(self):
        pass

# Concrete observer
class AnalogClock(Observer):
    def on_update(self):
        pass

timer = ClockTimer()
digital_clock = DigitalClock(timer)
analog_clock = AnalogClock(timer)
```

The two clock objects will be notified whenever the timer is “ticked”, though neither the timer nor the clocks needed prior knowledge of the other. During object construction any relationships between subjects and observers can be created.

Documenting events

pyglet uses a modified version of *Epydoc* to construct its API documentation. One of these modifications is the inclusion of an “Events” summary for event dispatchers. If you plan on releasing your code as a library for others to use, you may want to consider using the same tool to document code.

The patched version of *Epydoc* is included in the pyglet repository under `trunk/tools/epydoc` (it is not included in distributions). It has special notation for document event methods, and allows conditional execution when introspecting source code.

If the `sys.is_epydoc` attribute exists and is `True`, the module is currently being introspected for documentation. pyglet places event documentation only within this conditional, to prevent extraneous methods appearing on the class.

To document an event, create a method with the event’s signature and add a blank `event` field to the docstring:

```
import sys

class MyDispatcher(object):
    if getattr(sys, 'is_epydoc'):
        def on_update():
            '''The object was updated.

            :event:
            '''
```

Note that the event parameters should not include `self`. The function will appear in the “Events” table and not as a method.

Working with the keyboard

pyglet has support for low-level keyboard input suitable for games as well as locale- and device-independent Unicode text entry.

Keyboard input requires a window which has focus. The operating system usually decides which application window has keyboard focus. Typically this window appears above all others and may be decorated differently, though this is platform-specific (for example, Unix window managers sometimes couple keyboard focus with the mouse pointer).

You can request keyboard focus for a window with the *activate* method, but you should not rely on this – it may simply provide a visual cue to the user indicating that the window requires user input, without actually getting focus.

Windows created with the *WINDOW_STYLE_BORDERLESS* or *WINDOW_STYLE_TOOL* style cannot receive keyboard focus.

It is not possible to use pyglet’s keyboard or text events without a window; consider using Python built-in functions such as `raw_input` instead.

- *Keyboard events*
 - *Defined key symbols*
 - *Modifiers*
 - *User-defined key symbols*
 - *Remembering key state*
- *Text and motion events*
 - *Motion events*
- *Keyboard exclusivity*

Keyboard events

The *Window.on_key_press* and *Window.on_key_release* events are fired when any key on the keyboard is pressed or released, respectively. These events are not affected by “key repeat” – once a key is pressed there are no more events for that key until it is released.

Both events are parameterised by the same arguments:

```
def on_key_press(symbol, modifiers):
    pass
```



```
def on_key_release(symbol, modifiers):
    pass
```

Defined key symbols

The *symbol* argument is an integer that represents a “virtual” key code. It does *not* correspond to any particular numbering scheme; in particular the symbol is *not* an ASCII character code.

pyglet has key symbols that are hardware and platform independent for many types of keyboard. These are defined in *pyglet.window.key* as constants. For example, the Latin-1 alphabet is simply the letter itself:

```
key.A
key.B
key.C
...
```

The numeric keys have an underscore to make them valid identifiers:

```
key._1
key._2
key._3
...
```

Various control and directional keys are identified by name:

```
key.ENTER or key.RETURN
key.SPACE
key.BACKSPACE
key.DELETE
key.MINUS
key.EQUAL
key.BACKSLASH

key.LEFT
key.RIGHT
key.UP
key.DOWN
key.HOME
key.END
key.PAGEUP
key.PAGEDOWN

key.F1
key.F2
...
```

Keys on the number pad have separate symbols:

```
key.NUM_1
key.NUM_2
...
key.NUM_EQUAL
key.NUM_DIVIDE
key.NUM_MULTIPLY
key.NUM_SUBTRACT
key.NUM_ADD
key.NUM_DECIMAL
key.NUM_ENTER
```

Some modifier keys have separate symbols for their left and right sides (however they cannot all be distinguished on all platforms, including Mac OS X):

```
key.LCTRL
key.RCTRL
key.LSHIFT
key.RSHIFT
...
```

Key symbols are independent of any modifiers being held down. For example, lower-case and upper-case letters both generate the *A* symbol. This is also true of the number keypad.

Modifiers

The modifiers that are held down when the event is generated are combined in a bitwise fashion and provided in the `modifiers` parameter. The modifier constants defined in `pyglet.window.key` are:

```
MOD_SHIFT
MOD_CTRL
MOD_ALT           Not available on Mac OS X
MOD_WINDOWS       Available on Windows only
MOD_COMMAND       Available on Mac OS X only
MOD_OPTION        Available on Mac OS X only
MOD_CAPSLOCK
MOD_NUMLOCK
MOD_SCROLLLOCK
MOD_ACCEL         Equivalent to MOD_CTRL, or MOD_COMMAND on Mac OS X.
```

For example, to test if the shift key is held down:

```
if modifiers & MOD_SHIFT:
    pass
```

Unlike the corresponding key symbols, it is not possible to determine whether the left or right modifier is held down (though you could emulate this behaviour by keeping track of the key states yourself).

User-defined key symbols

pyglet does not define key symbols for every keyboard ever made. For example, non-Latin languages will have many keys not recognised by pyglet (however, their Unicode representation will still be valid, see *Text and motion events*). Even English keyboards often have additional so-called “OEM” keys added by the manufacturer, which might be labelled “Media”, “Volume” or “Shopping”, for example.

In these cases pyglet will create a key symbol at runtime based on the hardware scancode of the key. This is guaranteed to be unique for that model of keyboard, but may not be consistent across other keyboards with the same labelled key.

The best way to use these keys is to record what the user presses after a prompt, and then check for that same key symbol. Many commercial games have similar functionality in allowing players to set up their own key bindings.

Remembering key state

pyglet provides the convenience class *KeyStateHandler* for storing the current keyboard state. This can be pushed onto the event handler stack of any window and subsequently queried as a dict:

```

from pyglet.window import key

window = pyglet.window.Window()
keys = key.KeyStateHandler()
window.push_handlers(keys)

# Check if the spacebar is currently pressed:
if keys[key.SPACE]:
    pass

```

Text and motion events

pyglet decouples the keys that the user presses from the Unicode text that is input. There are several benefits to this:

- The complex task of mapping modifiers and key symbols to Unicode characters is taken care of automatically and correctly.
- Key repeat is applied to keys held down according to the user's operating system preferences.
- Dead keys and compose keys are automatically interpreted to produce diacritic marks or combining characters.
- Keyboard input can be routed via an input palette, for example to input characters from Asian languages.
- Text input can come from other user-defined sources, such as handwriting or voice recognition.

The actual source of input (i.e., which keys were pressed, or what input method was used) should be considered outside of the scope of the application – the operating system provides the necessary services.

When text is entered into a window, the *on_text* event is fired:

```

def on_text(text):
    pass

```

The only parameter provided is a Unicode string. For keyboard input this will usually be one character long, however more complex input methods such as an input palette may provide an entire word or phrase at once.

You should always use the *on_text* event when you need to determine a string from a sequence of keystrokes. Conversely, you never use *on_text* when you require keys to be pressed (for example, to control the movement of the player in a game).

Motion events

In addition to entering text, users press keys on the keyboard to navigate around text widgets according to well-ingrained conventions. For example, pressing the left arrow key moves the cursor one character to the left.

While you might be tempted to use the *on_key_press* event to capture these events, there are a couple of problems:

- Key repeat events are not generated for *on_key_press*, yet users expect that holding down the left arrow key will eventually move the character to the beginning of the line.
- Different operating systems have different conventions for the behaviour of keys. For example, on Windows it is customary for the Home key to move the cursor to the beginning of the line, whereas on Mac OS X the same key moves to the beginning of the document.

pyglet windows provide the *on_text_motion* event, which takes care of these problems by abstracting away the key presses and providing your application only with the intended cursor motion:

```

def on_text_motion(motion):
    pass

```

motion is an integer which is a constant defined in *pyglet.window.key*. The following table shows the defined text motions and their keyboard mapping on each operating system.

Constant	Behaviour	Win-dows/Linux	Mac OS X
MOTION_UP	Move the cursor up	Up	Up
MOTION_DOWN	Move the cursor down	Down	Down
MOTION_LEFT	Move the cursor left	Left	Left
MOTION_RIGHT	Move the cursor right	Right	Right
MOTION_PREVIOUS_WORD	Move the cursor to the previous word	Ctrl + Left	Option + Left
MOTION_NEXT_WORD	Move the cursor to the next word	Ctrl + Right	Option + Right
MOTION_BEGINNING_OF_LINE	Move the cursor to the beginning of the current line	Home	Command + Left
MOTION_END_OF_LINE	Move the cursor to the end of the current line	End	Command + Right
MOTION_PREVIOUS_PAGE	Move to the previous page	Page Up	Page Up
MOTION_NEXT_PAGE	Move to the next page	Page Down	Page Down
MOTION_BEGINNING_OF_DOCUMENT	Move to the beginning of the document	Ctrl + Home	Home
MOTION_END_OF_FILE	Move to the end of the document	Ctrl + End	End
MOTION_BACKSPACE	Delete the previous character	Backspace	Backspace
MOTION_DELETE	Delete the next character, or the current character	Delete	Delete

Keyboard exclusivity

Some keystrokes or key combinations normally bypass applications and are handled by the operating system. Some examples are Alt+Tab (Command+Tab on Mac OS X) to switch applications and the keys mapped to Expose on Mac OS X.

You can disable these hot keys and have them behave as ordinary keystrokes for your application. This can be useful if you are developing a kiosk application which should not be closed, or a game in which it is possible for a user to accidentally press one of these keys.

To enable this mode, call *set_exclusive_keyboard* for the window on which it should apply. On Mac OS X the dock and menu bar will slide out of view while exclusive keyboard is activated.

The following restrictions apply on Windows:

- Most keys are not disabled: a user can still switch away from your application using Ctrl+Escape, Alt+Escape, the Windows key or Ctrl+Alt+Delete. Only the Alt+Tab combination is disabled.

The following restrictions apply on Mac OS X:

- The power key is not disabled.

Use of this function is not recommended for general release applications or games as it violates user-interface conventions.

Working with the mouse

All pyglet windows can receive input from a 3 button mouse with a 2 dimensional scroll wheel. The mouse pointer is typically drawn by the operating system, but you can override this and request either a different cursor shape or provide your own image or animation.

- *Mouse events*
- *Changing the mouse cursor*
- *Mouse exclusivity*

Mouse events

All mouse events are dispatched by the window which receives the event from the operating system. Typically this is the window over which the mouse cursor is, however mouse exclusivity and drag operations mean this is not always the case.

The coordinate space for the mouse pointer's location is relative to the bottom-left corner of the window, with increasing Y values approaching the top of the screen (note that this is “upside-down” compared with many other windowing toolkits, but is consistent with the default OpenGL projection in pyglet).

Fig. 1.4: The coordinate space for the mouse pointer.

The most basic mouse event is *on_mouse_motion* which is dispatched every time the mouse moves:

```
def on_mouse_motion(x, y, dx, dy):
    pass
```

The *x* and *y* parameters give the coordinates of the mouse pointer, relative to the bottom-left corner of the window.

The event is dispatched every time the operating system registers a mouse movement. This is not necessarily once for every pixel moved – the operating system typically samples the mouse at a fixed frequency, and it is easy to move the mouse faster than this. Conversely, if your application is not processing events fast enough you may find that several queued-up mouse events are dispatched in a single *Window.dispatch_events* call. There is no need to concern yourself with either of these issues; the latter rarely causes problems, and the former can not be avoided.

Many games are not concerned with the actual position of the mouse cursor, and only need to know in which direction the mouse has moved. For example, the mouse in a first-person game typically controls the direction the player looks, but the mouse pointer itself is not displayed.

The *dx* and *dy* parameters are for this purpose: they give the distance the mouse travelled along each axis to get to its present position. This can be computed naively by storing the previous *x* and *y* parameters after every mouse event, but besides being tiresome to code, it does not take into account the effects of other obscuring windows. It is best to use the *dx* and *dy* parameters instead.

The following events are dispatched when a mouse button is pressed or released, or the mouse is moved while any button is held down:

```
def on_mouse_press(x, y, button, modifiers):
    pass

def on_mouse_release(x, y, button, modifiers):
    pass

def on_mouse_drag(x, y, dx, dy, buttons, modifiers):
    pass
```

The *x*, *y*, *dx* and *dy* parameters are as for the *on_mouse_motion* event. The press and release events do not require *dx* and *dy* parameters as they would be zero in this case. The *modifiers* parameter is as for the keyboard events, see *Working with the keyboard*.

The *button* parameter signifies which mouse button was pressed, and is one of the following constants:

```
pyglet.window.mouse.LEFT
pyglet.window.mouse.MIDDLE
pyglet.window.mouse.RIGHT
```

The *buttons* parameter in *on_mouse_drag* is a bitwise combination of all the mouse buttons currently held down. For example, to test if the user is performing a drag gesture with the left button:

```
from pyglet.window import mouse

def on_mouse_drag(x, y, dx, dy, buttons, modifiers):
    if buttons & mouse.LEFT:
        pass
```

When the user begins a drag operation (i.e., pressing and holding a mouse button and then moving the mouse), the window in which they began the drag will continue to receive the *on_mouse_drag* event as long as the button is held down. This is true even if the mouse leaves the window. You generally do not need to handle this specially: it is a convention among all operating systems that dragging is a gesture rather than a direct manipulation of the user interface widget.

There are events for when the mouse enters or leaves a window:

```
def on_mouse_enter(x, y):
    pass

def on_mouse_leave(x, y):
    pass
```

The coordinates for *on_mouse_leave* will lie outside of your window. These events are not dispatched while a drag operation is taking place.

The mouse scroll wheel generates the *on_mouse_scroll* event:

```
def on_mouse_scroll(x, y, scroll_x, scroll_y):
    pass
```

The *scroll_y* parameter gives the number of “clicks” the wheel moved, with positive numbers indicating the wheel was pushed forward. The *scroll_x* parameter is 0 for most mice, however some new mice such as the Apple Mighty Mouse use a ball instead of a wheel; the *scroll_x* parameter gives the horizontal movement in this case. The scale of these numbers is not known; it is typically set by the user in their operating system preferences.

Changing the mouse cursor

The mouse cursor can be set to one of the operating system cursors, a custom image, or hidden completely. The change to the cursor will be applicable only to the window you make the change to. To hide the mouse cursor, call *Window.set_mouse_visible*:







































```
window = pyglet.window.Window()
window.set_mouse_visible(False)
```

This can be useful if the mouse would obscure text that the user is typing. If you are hiding the mouse cursor for use in a game environment, consider making the mouse exclusive instead; see *Mouse exclusivity*, below.

Use *Window.set_mouse_cursor* to change the appearance of the mouse cursor. A mouse cursor is an instance of *MouseCursor*. You can obtain the operating system-defined cursors with *Window.get_system_mouse_cursor*:

```
cursor = window.get_system_mouse_cursor(win.CURSOR_HELP)
window.set_mouse_cursor(cursor)
```

The cursors that pyglet defines are listed below, along with their typical appearance on Windows and Mac OS X. The pointer image on Linux is dependent on the window manager.

Constant	Windows XP	Mac OS X
<i>CURSOR_DEFAULT</i>		
<i>CURSOR_CROSSHAIR</i>		
<i>CURSOR_HAND</i>		
<i>CURSOR_HELP</i>		
<i>CURSOR_NO</i>		
<i>CURSOR_SIZE</i>		
<i>CURSOR_SIZE_DOWN</i>		
<i>CURSOR_SIZE_DOWN_LEFT</i>		
<i>CURSOR_SIZE_DOWN_RIGHT</i>		
<i>CURSOR_SIZE_LEFT</i>		
<i>CURSOR_SIZE_LEFT_RIGHT</i>		
<i>CURSOR_SIZE_RIGHT</i>		
<i>CURSOR_SIZE_UP</i>		
<i>CURSOR_SIZE_UP_DOWN</i>		
<i>CURSOR_SIZE_UP_LEFT</i>		
<i>CURSOR_SIZE_UP_RIGHT</i>		
<i>CURSOR_TEXT</i>		
<i>CURSOR_WAIT</i>		
<i>CURSOR_WAIT_ARROW</i>		

Alternatively, you can use your own image as the mouse cursor. Use `pyglet.image.load` to load the image, then create an `ImageMouseCursor` with the image and “hot-spot” of the cursor. The hot-spot is the point of the image that corresponds to the actual pointer location on screen, for example, the point of the arrow:

```
image = pyglet.image.load('cursor.png')
cursor = pyglet.window.ImageMouseCursor(image, 16, 8)
window.set_mouse_cursor(cursor)
```

You can even render a mouse cursor directly with OpenGL. You could draw a 3-dimensional cursor, or a particle trail, for example. To do this, subclass `MouseCursor` and implement your own draw method. The draw method will be called with the default pyglet window projection, even if you are using another projection in the rest of your application.

Mouse exclusivity

It is possible to take complete control of the mouse for your own application, preventing it being used to activate other applications. This is most useful for immersive games such as first-person shooters.

When you enable mouse-exclusive mode, the mouse cursor is no longer available. It is not merely hidden – no amount of mouse movement will make it leave your application. Because there is no longer a mouse cursor, the *x* and *y* parameters of the mouse events are meaningless; you should use only the *dx* and *dy* parameters to determine how the mouse was moved.

Activate mouse exclusive mode with *set_exclusive_mouse*:

```
window = pyglet.window.Window()
window.set_exclusive_mouse(True)
```

You should activate mouse exclusive mode even if your window is full-screen: it will prevent the window “hitting” the edges of the screen, and behave correctly in multi-monitor setups (a common problem with commercial full-screen games is that the mouse is only hidden, meaning it can accidentally travel onto the other monitor where applications are still visible).

Note that on Linux setting exclusive mouse also disables Alt+Tab and other hotkeys for switching applications. No workaround for this has yet been discovered.

Working with other input devices

Pyglet’s *input* module allows you to accept input from any USB human interface device (HID). High level interfaces are provided for working with joysticks and with the Apple Remote.

- *Using joysticks*
- *Using the Apple Remote*

Using joysticks

Before using a joystick, you must find it and open it. To get a list of all joystick devices currently connected to your computer, call *pyglet.input.get_joysticks*:

```
joysticks = pyglet.input.get_joysticks()
```

Then choose a joystick from the list and call *Joystick.open* to open the device:

```
if joysticks:
    joystick = joysticks[0]
    joystick.open()
```

You may immediately begin querying the state of the joystick by looking at its attributes. The current position of the joystick is recorded in its ‘x’ and ‘y’ attributes, both of which are normalized to values within the range of -1 to 1. For the x-axis, *x* = -1 means the joystick is pushed all the way to the left and *x* = 1 means the joystick is pushed to the right. For the y-axis, a value of *y* = -1 means that the joystick is pushed up and a value of *y* = 1 means that the joystick is pushed down.

If your joystick has two analog controllers, the position of the second controller is typically given by *z* and *rz*, where *z* is the horizontal axis position and *rz* is the vertical axis position.

The state of the joystick buttons is contained in the *buttons* attribute as a list of boolean values. A True value indicates that the corresponding button is being pressed. While buttons may be labeled A, B, X, or Y on the physical joystick,

they are simply referred to by their index when accessing the *buttons* list. There is no way to know which button index corresponds to which physical button on the device without simply testing the particular joystick. So it is a good idea to let users change button assignments.

Each open joystick dispatches events when the joystick changes state. For buttons, there is the *on_joybutton_press* event which is sent whenever any of the joystick's buttons are pressed:

```
def on_joybutton_press(joystick, button):
    pass
```

and the *on_joybutton_release* event which is sent whenever any of the joystick's buttons are released:

```
def on_joybutton_release(joystick, button):
    pass
```

The *joystick* parameter is the *Joystick* instance whose buttons changed state (useful if you have multiple joysticks connected). The *button* parameter signifies which button changed and is simply an integer value, the index of the corresponding button in the *buttons* list.

For most games, it is probably best to examine the current position of the joystick directly by using the *x* and *y* attributes. However if you want to receive notifications whenever these values change you should handle the *on_joyaxis_motion* event:

```
def on_joyaxis_motion(joystick, axis, value):
    pass
```

The *joystick* parameter again tells you which joystick device changed. The *axis* parameter is string such as "x", "y", or "rx" telling you which axis changed value. And *value* gives the current normalized value of the axis, ranging between -1 and 1.

If the joystick has a hat switch, you may examine its current value by looking at the *hat_x* and *hat_y* attributes. For both, the values are either -1, 0, or 1. Note that *hat_y* will output 1 in the up position and -1 in the down position, which is the opposite of the y-axis control.

To be notified when the hat switch changes value, handle the *on_joyhat_motion* event:

```
def on_joyhat_motion(joystick, hat_x, hat_y):
    pass
```

The *hat_x* and *hat_y* parameters give the same values as the joystick's *hat_x* and *hat_y* attributes.

A good way to use the joystick event handlers might be to define them within a controller class and then call:

```
joystick.push_handlers(my_controller)
```

Using the Apple Remote

The Apple Remote is a small infrared remote originally distributed with the iMac. The remote has six buttons, which are accessed with the names *left*, *right*, *up*, *down*, *menu*, and *select*. Additionally when certain buttons are held down, they act as virtual buttons. These are named *left_hold*, *right_hold*, *menu_hold*, and *select_hold*.

To use the remote, first call *get_apple_remote*:

```
remote = pyglet.input.get_apple_remote()
```

Then open it:

```
if remote:
    remote.open(window, exclusive=True)
```

The remote is opened in exclusive mode so that while we are using the remote in our program, pressing the buttons does not activate Front Row, or change the volume, etc. on the computer.

The following event handlers tell you when a button on the remote has been either pressed or released:

```
def on_button_press(button):
    pass

def on_button_release(button):
    pass
```

The *button* parameter indicates which button changed and is a string equal to one of the ten button names defined above: “up”, “down”, “left”, “left_hold”, “right”, “right_hold”, “select”, “select_hold”, “menu”, or “menu_hold”.

To use the remote, you may define code for the event handlers in some controller class and then call:

```
remote.push_handlers(my_controller)
```

Keeping track of time

pyglet’s *clock* module provides functionality for scheduling functions for periodic or one-shot future execution and for calculating and displaying the application frame rate.

- *Calling functions periodically*
- *Animation techniques*
- *The frame rate*
 - *Displaying the frame rate*
- *User-defined clocks*

Calling functions periodically

pyglet applications begin execution with:

```
pyglet.app.run()
```

Once called, this function doesn’t return until the application windows have been closed. This may leave you wondering how to execute code while the application is running.

Typical applications need to execute code in only three circumstances:

- A user input event (such as a mouse movement or key press) has been generated. In this case the appropriate code can be attached as an event handler to the window.
- An animation or other time-dependent system needs to update the position or parameters of an object. We’ll call this a “periodic” event.
- A certain amount of time has passed, perhaps indicating that an operation has timed out, or that a dialog can be automatically dismissed. We’ll call this a “one-shot” event.

To have a function called periodically, for example, once every 0.1 seconds:

```
def update(dt):
    # ...
pyglet.clock.schedule_interval(update, 0.1)
```

The *dt* parameter gives the number of seconds (due to latency, load and timer inprecision, this might be slightly more or less than the requested interval).

Scheduling functions with a set interval is ideal for animation, physics simulation, and game state updates. pyglet ensures that the application does not consume more resources than necessary to execute the scheduled functions in time.

Rather than “limiting the frame rate”, as required in other toolkits, simply schedule all your update functions for no less than the minimum period your application or game requires. For example, most games need not run at more than 60Hz (60 times a second) for imperceptibly smooth animation, so the interval given to *schedule_interval* would be $1/60.0$ (or more).

If you are writing a benchmarking program or otherwise wish to simply run at the highest possible frequency, use *schedule*:

```
def update(dt):
    # ...
pyglet.clock.schedule(update)
```

By default pyglet window buffer swaps are synchronised to the display refresh rate, so you may also want to disable *set_vsync*.

For one-shot events, use *schedule_once*:

```
def dismiss_dialog(dt):
    # ...

# Dismiss the dialog after 5 seconds.
pyglet.clock.schedule_once(dismiss_dialog, 5.0)
```

To stop a scheduled function from being called, including cancelling a periodic function, use *pyglet.clock.unschedule*.

Animation techniques

Every scheduled function takes a *dt* parameter, giving the actual “wall clock” time that passed since the previous invocation (or the time the function was scheduled, if it’s the first period). This parameter can be used for numerical integration.

For example, a non-accelerating particle with velocity *v* will travel some distance over a change in time *dt*. This distance is calculated as $v * dt$. Similarly, a particle under constant acceleration *a* will have a change in velocity of $a * dt$.

The following example demonstrates a simple way to move a sprite across the screen at exactly 10 pixels per second:

```
sprite = pyglet.sprite.Sprite(image)
sprite.dx = 10.0

def update(dt):
    sprite.x += sprite.dx * dt
pyglet.clock.schedule_interval(update, 1/60.0) # update at 60Hz
```

This is a robust technique for simple animation, as the velocity will remain constant regardless of the speed or load of the computer.

Some examples of other common animation variables are given in the table below.

Animation parameter	Distance	Velocity
Rotation	Degrees	Degrees per second
Position	Pixels	Pixels per second
Keyframes	Frame number	Frames per second

The frame rate

Game performance is often measured in terms of the number of times the display is updated every second; that is, the frames-per-second or FPS. You can determine your application's FPS with a single function call:

```
pyglet.clock.get_fps()
```

The value returned is more useful than simply taking the reciprocal of dt from a period function, as it is averaged over a sliding window of several frames.

Displaying the frame rate

A simple way to profile your application performance is to display the frame rate while it is running. Printing it to the console is not ideal as this will have a severe impact on performance. pyglet provides the *ClockDisplay* class for displaying the frame rate with very little effort:

```
fps_display = pyglet.clock.ClockDisplay()

@window.event
def on_draw():
    window.clear()
    fps_display.draw()
```

By default the frame rate will be drawn in the bottom-right corner of the window in a semi-translucent large font. See the *ClockDisplay* documentation for details on how to customise this, or even display another clock value (such as the current time) altogether.

User-defined clocks

The default clock used by pyglet uses the system clock to determine the time (i.e., `time.time()`). Separate clocks can be created, however, allowing you to use another time source. This can be useful for implementing a separate “game time” to the real-world time, or for synchronising to a network time source or a sound device.

Each of the *clock* functions are aliases for the methods on a global instance of *clock.Clock*. You can construct or subclass your own *Clock*, which can then maintain its own schedule and framerate calculation. See the class documentation for more details.

Displaying text

pyglet provides the *font* module for rendering high-quality antialiased Unicode glyphs efficiently. Any installed font on the operating system is seen by pyglet, or you can supply your own font with your application.

Notice that not all font formats are supported, see [Supported font formats](#)

Text rendering is performed with the *text* module, which can display word-wrapped formatted text. There is also support for interactive editing of text on-screen with a caret.

- *Simple text rendering*
- *The document/layout model*
 - *Documents*
 - *Layouts*
- *Formatted text*
 - *Character styles*
 - *Paragraph styles*
 - * *Tabs*
 - *Attributed text*
 - *HTML*
- *Custom elements*
- *User-editable text*
- *Loading system fonts*
- *Font sizes*
 - *Font resolution*
 - *Determining font size*
- *Loading custom fonts*
 - *Supported font formats*
- *OpenGL font considerations*
 - *Context affinity*
 - *Blend state*

Simple text rendering

The following complete example creates a window that displays “Hello, World” centered vertically and horizontally:

```

window = pyglet.window.Window()
label = pyglet.text.Label('Hello, world',
                           font_name='Times New Roman',
                           font_size=36,
                           x=window.width//2, y=window.height//2,
                           anchor_x='center', anchor_y='center')

@window.event
def on_draw():
    window.clear()
    label.draw()

pyglet.app.run()
```

The example demonstrates the most common uses of text rendering:

- The font name and size are specified directly in the constructor. Additional parameters exist for setting the bold and italic styles and the color of the text.
- The position of the text is given by the *x* and *y* coordinates. The meaning of these coordinates is given by the *anchor_x* and *anchor_y* parameters.
- The actual text is drawn with the *Label.draw* method. Labels can also be added to a graphics batch; see *Graphics* for details.

The *HTMLLabel* class is used similarly, but accepts an HTML formatted string instead of parameters describing the style. This allows the label to display text with mixed style:

```

label = pyglet.text.HTMLLabel(
    '<font face="Times New Roman" size="4">Hello, <i>world</i></font>',
```

```
x=window.width//2, y=window.height//2,  
anchor_x='center', anchor_y='center')
```

See *Formatted text* for details on the subset of HTML that is supported.

The document/layout model

The *Label* class demonstrated above presents a simplified interface to pyglet’s complete text rendering capabilities. The underlying *TextLayout* and *AbstractDocument* classes provide a “model/view” interface to all of pyglet’s text features.

Documents

A *document* is the “model” part of the architecture, and describes the content and style of the text to be displayed. There are two concrete document classes: *UnformattedDocument* and *FormattedDocument*. *UnformattedDocument* models a document containing text in just one style, whereas *FormattedDocument* allows the style to change within the text.

An empty, unstyled document can be created by constructing either of the classes directly. Usually you will want to initialise the document with some text, however. The *decode_text*, *decode_attributed* and *decode_html* functions return a document given a source string. For *decode_text*, this is simply a plain text string, and the return value is an *UnformattedDocument*:

```
document = pyglet.text.decode_text('Hello, world.')
```

decode_attributed and *decode_html* are described in detail in the next section.

The text of a document can be modified directly as a property on the object:

```
document.text = 'Goodbye, cruel world.'
```

However, if small changes are being made to the document it can be more efficient (when coupled with an appropriate layout; see below) to use the *delete_text* and *insert_text* methods instead.

Layouts

The actual layout and rendering of a document is performed by the *TextLayout* classes. This split exists to reduce the complexity of the code, and to allow a single document to be displayed in multiple layouts simultaneously (in other words, many layouts can display one document).

Each of the *TextLayout* classes perform layout in the same way, but represent a trade-off in efficiency of update against efficiency of drawing and memory usage.

The base *TextLayout* class uses little memory, and shares its graphics group with other *TextLayout* instances in the same batch (see *Batched rendering*). When the text or style of the document is modified, or the layout constraints change (for example, the width of the layout changes), the entire text layout is recalculated. This is a potentially expensive operation, especially for long documents. This makes *TextLayout* suitable for relatively short or unchanging documents.

ScrollableTextLayout is a small extension to *TextLayout* that clips the text to a specified view rectangle, and allows text to be scrolled within that rectangle without performing the layout calculation again. Because of this clipping rectangle the graphics group cannot be shared with other text layouts, so for ideal performance *ScrollableTextLayout* should be used only if this behaviour is required.

IncrementalTextLayout uses a more sophisticated layout algorithm that performs less work for small changes to documents. For example, if a document is being edited by the user, only the immediately affected lines of text are recalculated when a character is typed or deleted. *IncrementalTextLayout* also performs view rectangle culling, reducing the amount of layout and rendering required when the document is larger than the view. *IncrementalTextLayout* should be used for large documents or documents that change rapidly.

All the layout classes can be constructed given a document and display dimensions:

```
layout = pyglet.text.layout.TextLayout(document, width, height)
```

Additional arguments to the constructor allow the specification of a graphics batch and group (recommended if many layouts are to be rendered), and the optional *multiline* and *wrap_lines* flags.

multiline To honor newlines in the document you will need to set this to `True`. If you do not then newlines will be rendered as plain spaces.

wrap_lines If you expect that your document lines will be wider than the display width then pyglet can automatically wrap them to fit the width by setting this option to `True`.

Like labels, layouts are positioned through their *x*, *y*, *anchor_x* and *anchor_y* properties. Note that unlike *AbstractImage*, the *anchor* properties accept a string such as "bottom" or "center" instead of a numeric displacement.

Formatted text

The *FormattedDocument* class maintains style information for individual characters in the text, rather than a single style for the whole document. Styles can be accessed and modified by name, for example:

```
# Get the font name used at character index 0
font_name = document.get_style('font_name', 0)

# Set the font name and size for the first 5 characters
document.set_style(0, 5, dict(font_name='Arial', font_size=12))
```

Internally, character styles are run-length encoded over the document text; so longer documents with few style changes do not use excessive memory.

From the document's point of view, there are no predefined style names: it simply maps names and character ranges to arbitrary Python values. It is the *TextLayout* classes that interpret this style information; for example, by selecting a different font based on the *font_name* style. Unrecognised style names are ignored by the layout – you can use this knowledge to store additional data alongside the document text (for example, a URL behind a hyperlink).

Character styles

The following character styles are recognised by all *TextLayout* classes.

Where an attribute is marked "as a distance" the value is assumed to be in pixels if given as an int or float, otherwise a string of the form "0u" is required, where 0 is the distance and u is the unit; one of "px" (pixels), "pt" (points), "pc" (picas), "cm" (centimeters), "mm" (millimeters) or "in" (inches). For example, "14pt" is the distance covering 14 points, which at the default DPI of 96 is 18 pixels.

font_name Font family name, as given to *pyglet.font.load*.

font_size Font size, in points.

bold Boolean.

italic Boolean.

underline 4-tuple of ints in range (0, 255) giving RGBA underline color, or None (default) for no underline.

kerning Additional space to insert between glyphs, as a distance. Defaults to 0.

baseline Offset of glyph baseline from line baseline, as a distance. Positive values give a superscript, negative values give a subscript. Defaults to 0.

color 4-tuple of ints in range (0, 255) giving RGBA text color

background_color 4-tuple of ints in range (0, 255) giving RGBA text background color; or `None` for no background fill.

Paragraph styles

Although *FormattedDocument* does not distinguish between character- and paragraph-level styles, *TextLayout* interprets the following styles only at the paragraph level. You should take care to set these styles for complete paragraphs only, for example, by using *FormattedDocument.set_paragraph_style*.

These styles are ignored for layouts without the `multiline` flag set.

align "left" (default), "center" or "right".

indent Additional horizontal space to insert before the first glyph of the first line of a paragraph, as a distance.

leading Additional space to insert between consecutive lines within a paragraph, as a distance. Defaults to 0.

line_spacing Distance between consecutive baselines in a paragraph, as a distance. Defaults to `None`, which automatically calculates the tightest line spacing for each line based on the maximum font ascent and descent.

margin_left Left paragraph margin, as a distance.

margin_right Right paragraph margin, as a distance.

margin_top Margin above paragraph, as a distance.

margin_bottom Margin below paragraph, as a distance. Adjacent margins do not collapse.

tab_stops List of horizontal tab stops, as distances, measured from the left edge of the text layout. Defaults to the empty list. When the tab stops are exhausted, they implicitly continue at 50 pixel intervals.

wrap Boolean. If `True` (the default), text wraps within the width of the layout.

For the purposes of these attributes, paragraphs are split by the newline character (U+0010) or the paragraph break character (U+2029). Line breaks within a paragraph can be forced with character U+2028.

Tabs A tab character in pyglet text is interpreted as ‘move to the next tab stop’. Tab stops are specified in pixels, not in some font unit; by default there is a tab stop every 50 pixels and because of that a tab can look too small for big fonts or too big for small fonts.

Additionally, when rendering text with tabs using a *monospace* font, character boxes may not align vertically.

To avoid these visualization issues the simpler solution is to convert the tabs to spaces before sending a string to a pyglet text-related class.

Attributed text

pyglet provides two formats for decoding formatted documents from plain text. These are useful for loading preprepared documents such as help screens. At this time there is no facility for saving (encoding) formatted documents.

The *attributed text* format is an encoding specific to pyglet that can exactly describe any *FormattedDocument*. You must use this encoding to access all of the features of pyglet text layout. For a more accessible, yet less featureful encoding, see the *HTML* encoding, described below.

The following example shows a simple attributed text encoded document:

```
Chapter 1

My father's family name being Pirrip, and my Christian name Philip,
my infant tongue could make of both names nothing longer or more
explicit than Pip. So, I called myself Pip, and came to be called
Pip.

I give Pirrip as my father's family name, on the authority of his
tombstone and my sister - Mrs. Joe Gargery, who married the
blacksmith. As I never saw my father or my mother, and never saw
any likeness of either of them (for their days were long before the
days of photographs), my first fancies regarding what they were
like, were unreasonably derived from their tombstones.
```

Newlines are ignored, unless two are made in succession, indicating a paragraph break. Line breaks can be forced with the `\\` sequence:

```
This is the way the world ends \\
This is the way the world ends \\
This is the way the world ends \\
Not with a bang but a whimper.
```

Line breaks are also forced when the text is indented with one or more spaces or tabs, which is useful for typesetting code:

```
The following paragraph has hard line breaks for every line of code:

import pyglet

window = pyglet.window.Window()
pyglet.app.run()
```

Text can be styled using a attribute tag:

```
This sentence makes a {bold True}bold{bold False} statement.
```

The attribute tag consists of the attribute name (in this example, `bold`) followed by a Python bool, int, float, string, tuple or list.

Unlike most structured documents such as HTML, attributed text has no concept of the “end” of a style; styles merely change within the document. This corresponds exactly to the representation used by *FormattedDocument* internally.

Some more examples follow:

```
{font_name 'Times New Roman'}{font_size 28>Hello{font_size 12},
{color (255, 0, 0, 255)}world{color (0, 0, 0, 255)}!
```

(This example uses 28pt Times New Roman for the word “Hello”, and 12pt red text for the word “world”).

Paragraph styles can be set by prefixing the style name with a period (.). This ensures the style range exactly encompasses the paragraph:

```
{.margin_left "12px"}This is a block quote, as the margin is inset.

{.margin_left "24px"}This paragraph is inset yet again.
```

Attributed text can be loaded as a Unicode string. In addition, any character can be inserted given its Unicode code point in numeric form, either in decimal:

```
This text is Copyright {#169}.
```

or hexadecimal:

```
This text is Copyright {#xa9}.
```

The characters { and } can be escaped by duplicating them:

```
Attributed text uses many "{{" and "}}" characters.
```

Use the `decode_attributed` function to decode attributed text into a *FormattedDocument*:

```
document = pyglet.text.decode_attributed('Hello, {bold True}world')
```

HTML

While attributed text gives access to all of the features of *FormattedDocument* and *TextLayout*, it is quite verbose and difficult produce text in. For convenience, pyglet provides an HTML 4.01 decoder that can translate a small, commonly used subset of HTML into a *FormattedDocument*.

Note that the decoder does not preserve the structure of the HTML document – all notion of element hierarchy is lost in the translation, and only the visible style changes are preserved.

The following example uses `decode_html` to create a *FormattedDocument* from a string of HTML:

```
document = pyglet.text.decode_html('Hello, <b>world</b>')
```

The following elements are supported:

```
B BLOCKQUOTE BR CENTER CODE DD DIR DL EM FONT H1 H2 H3 H4 H5 H6 I IMG KBD
LI MENU OL P PRE Q SAMP STRONG SUB SUP TT U UL VAR
```

The `style` attribute is not supported, so font sizes must be given as HTML logical sizes in the range 1 to 7, rather than as point sizes. The corresponding font sizes, and some other stylesheet parameters, can be modified by subclassing *HTMLDecoder*.

Custom elements

Graphics and other visual elements can be inserted inline into a document using *AbstractDocument.insert_element*. For example, inline elements are used to render HTML images included with the `IMG` tag. There is currently no support for floating or absolutely-positioned elements.

Elements must subclass *InlineElement* and override the `place` and `remove` methods. These methods are called by *TextLayout* when the element becomes or ceases to be visible. For *TextLayout* and *ScrollableTextLayout*, this is when the element is added or removed from the document; but for *IncrementalTextLayout* the methods are also called as the element scrolls in and out of the viewport.

The constructor of *InlineElement* gives the width and height (separated into the ascent above the baseline, and descent below the baseline) of the element.

Typically an *InlineElement* subclass will add graphics primitives to the layout's graphics batch; though applications may choose to simply record the position of the element and render it separately.

The position of the element in the document text is marked with a NUL character (U+0000) placeholder. This has the effect that inserting an element into a document increases the length of the document text by one. Elements can also be styled as if they were ordinary character text, though the layout ignores any such style attributes.

User-editable text

While pyglet does not come with any complete GUI widgets for applications to use, it does implement many of the features required to implement interactive text editing. These can be used as a basis for a more complete GUI system, or to present a simple text entry field, as demonstrated in the `examples/text_input.py` example.

IncrementalTextLayout should always be used for text that can be edited by the user. This class maintains information about the placement of glyphs on screen, and so can map window coordinates to a document position and vice-versa. These methods are *get_position_from_point*, *get_point_from_position*, *get_line_from_point*, *get_point_from_line*, *get_line_from_position*, *get_position_from_line*, *get_position_on_line* and *get_line_count*.

The viewable rectangle of the document can be adjusted using a document position instead of a scrollbar using the *ensure_line_visible* and *ensure_x_visible* methods.

IncrementalTextLayout can display a current text selection by temporarily overriding the foreground and background colour of the selected text. The *selection_start* and *selection_end* properties give the range of the selection, and *selection_color* and *selection_background_color* the colors to use (defaulting to white on blue).

The *Caret* class implements an insertion caret (cursor) for *IncrementalTextLayout*. This includes displaying the blinking caret at the correct location, and handling keyboard, text and mouse events. The behaviour in response to the events is very similar to the system GUIs on Windows, Mac OS X and GTK. Using *Caret* frees you from using the *IncrementalTextLayout* methods described above directly.

The following example creates a document, a layout and a caret and attaches the caret to the window to listen for events:

```
import pyglet

window = pyglet.window.Window()
document = pyglet.text.document.FormattedDocument()
layout = pyglet.text.layout.IncrementalTextLayout(document, width, height)
caret = pyglet.text.caret.Caret(layout)
window.push_handlers(caret)
```

When the layout is drawn, the caret will also be drawn, so this example is nearly complete enough to display the user input. However, it is suitable for use when only one editable text layout is to be in the window. If multiple text widgets are to be shown, some mechanism is needed to dispatch events to the widget that has keyboard focus. An example of how to do this is given in the *examples/text_input.py* example program.

Loading system fonts

The layout classes automatically load fonts as required. You can also explicitly load fonts to implement your own layout algorithms.

To load a font you must know its family name. This is the name displayed in the font dialog of any application. For example, all operating systems include the *Times New Roman* font. You must also specify the font size to load, in points:

```
# Load "Times New Roman" at 16pt
times = pyglet.font.load('Times New Roman', 16)
```

Bold and italic variants of the font can be specified with keyword parameters:

```
times_bold = pyglet.font.load('Times New Roman', 16, bold=True)
times_italic = pyglet.font.load('Times New Roman', 16, italic=True)
times_bold_italic = pyglet.font.load('Times New Roman', 16,
                                     bold=True, italic=True)
```

For maximum compatibility on all platforms, you can specify a list of font names to load, in order of preference. For example, many users will have installed the Microsoft Web Fonts pack, which includes *Verdana*, but this cannot be guaranteed, so you might specify *Arial* or *Helvetica* as suitable alternatives:

```
sans_serif = pyglet.font.load(('Verdana', 'Helvetica', 'Arial'), 16)
```

Also you can check for the availability of a font using *have_font*:

```
# Will return True
pyglet.font.have_font('Times New Roman')

# Will return False
pyglet.font.have_font('missing-font-name')
```

If you do not particularly care which font is used, and just need to display some readable text, you can specify *None* as the family name, which will load a default sans-serif font (Helvetica on Mac OS X, Arial on Windows XP):

```
sans_serif = pyglet.font.load(None, 16)
```

Font sizes

When loading a font you must specify the font size it is to be rendered at, in points. Points are a somewhat historical but conventional unit used in both display and print media. There are various conflicting definitions for the actual length of a point, but pyglet uses the PostScript definition: 1 point = 1/72 inches.

Font resolution

The actual rendered size of the font on screen depends on the display resolution. pyglet uses a default DPI of 96 on all operating systems. Most Mac OS X applications use a DPI of 72, so the font sizes will not match up on that operating system. However, application developers can be assured that font sizes remain consistent in pyglet across platforms.

The DPI can be specified directly in the *pyglet.font.load* function, and as an argument to the *TextLayout* constructor.

Determining font size

Once a font is loaded at a particular size, you can query its pixel size with the attributes:

```
Font.ascent
Font.descent
```

These measurements are shown in the diagram below.

Fig. 1.5: Font metrics. Note that the descent is usually negative as it descends below the baseline.

You can calculate the distance between successive lines of text as:

```
ascent - descent + leading
```

where *leading* is the number of pixels to insert between each line of text.

Loading custom fonts

You can supply a font with your application if it's not commonly installed on the target platform. You should ensure you have a license to distribute the font – the terms are often specified within the font file itself, and can be viewed with your operating system's font viewer.

Loading a custom font must be performed in two steps:

1. Let pyglet know about the additional font or font files.
2. Load the font by its family name.

For example, let's say you have the *Action Man* font in a file called `action_man.ttf`. The following code will load an instance of that font:

```
pyglet.font.add_file('action_man.ttf')
action_man = pyglet.font.load('Action Man')
```

Similarly, once the font file has been added, the font name can be specified as a style on a label or layout:

```
label = pyglet.text.Label('Hello', font_name='Action Man')
```

Fonts are often distributed in separate files for each variant. *Action Man Bold* would probably be distributed as a separate file called `action_man_bold.ttf`; you need to let pyglet know about this as well:

```
font.add_file('action_man_bold.ttf')
action_man_bold = font.load('Action Man', bold=True)
```

Note that even when you know the filename of the font you want to load, you must specify the font's family name to `pyglet.font.load`.

You need not have the file on disk to add it to pyglet; you can specify any file-like object supporting the `read` method. This can be useful for extracting fonts from a resource archive or over a network.

If the custom font is distributed with your application, consider using the *Application resources*.

Supported font formats

pyglet can load any font file that the operating system natively supports, but not all formats are fully supported.

The list of supported formats is shown in the table below.

Font Format	Windows XP	Mac OS X	Linux (FreeType)
TrueType (.ttf)	X	X	X
PostScript Type 1 (.pfm, .pfb)	X	X	X
Windows Bitmap (.fnt)	X		X
Mac OS X Data Fork Font (.dfont)		X	
OpenType (.otf) ¹		X	
X11 font formats PCF, BDF, SFONT			X
Bitstream PFR (.pfr)			X

Some of the fonts found in internet may miss information for some operating systems, others may have been written with work in progress tools not fully compliant with standards. Using the font with text editors or fonts viewers can help to determine if the font is broken.

¹ All OpenType fonts are backward compatible with TrueType, so while the advanced OpenType features can only be rendered with Mac OS X, the files can be used on any platform. pyglet does not currently make use of the additional kerning and ligature information within OpenType fonts. In Windows a few will use the variant `DEVICE_FONTTYPE` and may render bad, by example `inconsolata.otf`, from <http://levien.com/type/myfonts/inconsolata.html>

OpenGL font considerations

Text in pyglet is drawn using textured quads. Each font maintains a set of one or more textures, into which glyphs are uploaded as they are needed. For most applications this detail is transparent and unimportant, however some of the details of these glyph textures are described below for advanced users.

Context affinity

When a font is loaded, it immediately creates a texture in the current context's object space. Subsequent textures may need to be created if there is not enough room on the first texture for all the glyphs. This is done when the glyph is first requested.

pyglet always assumes that the object space that was active when the font was loaded is the active one when any texture operations are performed. Normally this assumption is valid, as pyglet shares object spaces between all contexts by default. There are a few situations in which this will not be the case, though:

- When explicitly setting the context share during context creation.
- When multiple display devices are being used which cannot support a shared context object space.

In any of these cases, you will need to reload the font for each object space that it's needed in. pyglet keeps a cache of fonts, but does so per-object-space, so it knows when it can reuse an existing font instance or if it needs to load it and create new textures. You will also need to ensure that an appropriate context is active when any glyphs may need to be added.

Blend state

The glyph textures have an internal format of `GL_ALPHA`, which provides a simple way to recolour and blend antialiased text by changing the vertex colors. pyglet makes very few assumptions about the OpenGL state, and will not alter it besides changing the currently bound texture.

The following blend state is used for drawing font glyphs:

```
from pyglet.gl import *
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
glEnable(GL_BLEND)
```

All glyph textures use the `GL_TEXTURE_2D` target, so you should ensure that a higher priority target such as `GL_TEXTURE_3D` is not enabled before trying to render text.

Images

pyglet provides functions for loading and saving images in various formats using native operating system services. pyglet can also work with the [Python Imaging Library](#) (PIL) for access to more file formats.

Loaded images can be efficiently provided to OpenGL as a texture, and OpenGL textures and framebuffers can be retrieved as pyglet images to be saved or otherwise manipulated.

pyglet also provides an efficient and comprehensive *Sprite* class, for displaying images on the screen with an optional transform.

- *Loading an image*
- *Supported image formats*
- *Working with images*
- *The AbstractImage hierarchy*
- *Accessing or providing pixel data*
 - *Performance concerns*
- *Image sequences and atlases*
 - *Image grids*
 - *3D textures*
 - *Texture bins and atlases*
- *Animations*
- *Buffer images*
- *Displaying images*
 - *Sprites*
 - *Simple image blitting*
- *OpenGL imaging*
 - *Texture dimensions*
 - *Texture internal format*
- *Saving an image*

Loading an image

Images can be loaded using the `pyglet.image.load` function:

```
kitten = pyglet.image.load('kitten.png')
```

If the image is distributed with your application, consider using the `pyglet.resource` module (see *Application resources*).

Without any additional arguments, `load` will attempt to load the filename specified using any available image decoder. This will allow you to load PNG, GIF, JPEG, BMP and DDS files, and possibly other files as well, depending on your operating system and additional installed modules (see the next section for details). If the image cannot be loaded, an `ImageDecodeException` will be raised.

You can load an image from any file-like object providing a `read` method by specifying the `file` keyword parameter:

```
kitten_stream = open('kitten.png', 'rb')
kitten = pyglet.image.load('kitten.png', file=kitten_stream)
```

In this case the filename `kitten.png` is optional, but gives a hint to the decoder as to the file type (it is otherwise unused).

pyglet provides the following image decoders:

Module	Class	Description
<code>pyglet.image.codecs.dds</code>	<code>DDSImageDecoder</code>	Reads Microsoft DirectDraw Surface files containing compressed textures
<code>pyglet.image.codecs.gdip</code>	<code>GDIPlusDecoder</code>	Uses Windows GDI+ services to decode images.
<code>pyglet.image.codecs.gtk</code>	<code>GdkPixbufImageDecoder</code>	Uses the GTK-2.0 GDK functions to decode images.
<code>pyglet.image.codecs.pil</code>	<code>PILImageDecoder</code>	Wrapper interface around PIL Image class.
<code>pyglet.image.codecs.png</code>	<code>PNGImageDecoder</code>	PNG decoder written in pure Python.
<code>pyglet.image.codecs.quicktime</code>	<code>QuickTimeImageDecoder</code>	Uses Mac OS X QuickTime to decode images.

Each of these classes registers itself with *pyglet.image* with the filename extensions it supports. The *load* function will try each image decoder with a matching file extension first, before attempting the other decoders. Only if every image decoder fails to load an image will *ImageDecodeException* be raised (the origin of the exception will be the first decoder that was attempted).

You can override this behaviour and specify a particular decoding instance to use. For example, in the following example the pure Python PNG decoder is always used rather than the operating system's decoder:

```
from pyglet.image.codecs.png import PNGImageDecoder
kitten = pyglet.image.load('kitten.png', decoder=PNGImageDecoder())
```

This use is not recommended unless your application has to work around specific deficiencies in an operating system decoder.

Supported image formats

The following table lists the image formats that can be loaded on each operating system. If PIL is installed, any additional formats it supports can also be read. See the [Python Imaging Library Handbook](#) for a list of such formats.

Extension	Description	Windows XP	Mac OS X	Linux ⁵
.bmp	Windows Bitmap	X	X	X
.dds	Microsoft DirectDraw Surface ⁶	X	X	X
.exif	Exif	X		
.gif	Graphics Interchange Format	X	X	X
.jpg .jpeg	JPEG/JIFF Image	X	X	X
.jp2 .jpx	JPEG 2000		X	
.pcx	PC Paintbrush Bitmap Graphic		X	
.png	Portable Network Graphic	X	X	X
.pnm	PBM Portable Any Map Graphic Bitmap			X
.ras	Sun raster graphic			X
.tga	Truevision Targa Graphic		X	
.tif .tiff	Tagged Image File Format	X	X	X
.xbm	X11 bitmap		X	X
.xpm	X11 icon		X	X

The only supported save format is PNG, unless PIL is installed, in which case any format it supports can be written.

Working with images

The *pyglet.image.load* function returns an *AbstractImage*. The actual class of the object depends on the decoder that was used, but all images support the following attributes:

width The width of the image, in pixels.

height The height of the image, in pixels.

anchor_x Distance of the anchor point from the left edge of the image, in pixels

anchor_y Distance of the anchor point from the bottom edge of the image, in pixels

⁵Requires GTK 2.0 or later.

⁶Only S3TC compressed surfaces are supported. Depth, volume and cube textures are not supported.

The anchor point defaults to (0, 0), though some image formats may contain an intrinsic anchor point. The anchor point is used to align the image to a point in space when drawing it.

You may only want to use a portion of the complete image. You can use the `get_region` method to return an image of a rectangular region of a source image:

```
image_part = kitten.get_region(x=10, y=10, width=100, height=100)
```

This returns an image with dimensions 100x100. The region extracted from *kitten* is aligned such that the bottom-left corner of the rectangle is 10 pixels from the left and 10 pixels from the bottom of the image.

Image regions can be used as if they were complete images. Note that changes to an image region may or may not be reflected on the source image, and changes to the source image may or may not be reflected on any region images. You should not assume either behaviour.

The AbstractImage hierarchy

The following sections deal with the various concrete image classes. All images subclass *AbstractImage*, which provides the basic interface described in previous sections.

Fig. 1.6: The *AbstractImage* class hierarchy.

An image of any class can be converted into a *Texture* or *ImageData* using the `get_texture` and `get_image_data` methods defined on *AbstractImage*. For example, to load an image and work with it as an OpenGL texture:

```
kitten = pyglet.image.load('kitten.png').get_texture()
```

There is no penalty for accessing one of these methods if object is already of the requested class. The following table shows how concrete classes are converted into other classes:

Original class	<code>.get_texture()</code>	<code>.get_image_data()</code>
<i>Texture</i>	No change	<code>glGetTexImage2D</code>
<i>TextureRegion</i>	No change	<code>glGetTexImage2D</code> , crop resulting image.
<i>ImageData</i>	<code>glTexImage2D</code> ¹	No change
<i>ImageDataRegion</i>	<code>glTexImage2D</code> ¹	No change
<i>CompressedImage-Data</i>	<code>glCompressedTexImage2D</code> ²	N/A ³
<i>BufferImage</i>	<code>glCopyTexSubImage2D</code> ⁴	<code>glReadPixels</code>

You should try to avoid conversions which use `glGetTexImage2D` or `glReadPixels`, as these can impose a substantial performance penalty by transferring data in the “wrong” direction of the video bus, especially on older hardware.

Accessing or providing pixel data

The *ImageData* class represents an image as a string or sequence of pixel data, or as a ctypes pointer. Details such as the pitch and component layout are also stored in the class. You can access an *ImageData* object for any image with `get_image_data`:

¹*ImageData* caches the texture for future use, so there is no performance penalty for repeatedly blitting an *ImageData*.

²If the required texture compression extension is not present, the image is decompressed in memory and then supplied to OpenGL via `glTexImage2D`.

³It is not currently possible to retrieve *ImageData* for compressed texture images. This feature may be implemented in a future release of pyglet. One workaround is to create a texture from the compressed image, then read the image data from the texture; i.e., `compressed_image.get_texture().get_image_data()`.

⁴*BufferImageMask* cannot be converted to *Texture*.

```
kitten = pyglet.image.load('kitten.png').get_image_data()
```

The design of *ImageData* is to allow applications to access the detail in the format they prefer, rather than having to understand the many formats that each operating system and OpenGL make use of.

The *pitch* and *format* properties determine how the bytes are arranged. *pitch* gives the number of bytes between each consecutive row. The data is assumed to run from left-to-right, bottom-to-top, unless *pitch* is negative, in which case it runs from left-to-right, top-to-bottom. There is no need for rows to be tightly packed; larger *pitch* values are often used to align each row to machine word boundaries.

The *format* property gives the number and order of color components. It is a string of one or more of the letters corresponding to the components in the following table:

R	Red
G	Green
B	Blue
A	Alpha
L	Luminance
I	Intensity

For example, a format string of "RGBA" corresponds to four bytes of colour data, in the order red, green, blue, alpha. Note that machine endianness has no impact on the interpretation of a format string.

The length of a format string always gives the number of bytes per pixel. So, the minimum absolute pitch for a given image is `len(kitten.format) * kitten.width`.

To retrieve pixel data in a particular format, use the *get_data* method, specifying the desired format and pitch. The following example reads tightly packed rows in RGB format (the alpha component, if any, will be discarded):

```
kitten = kitten.get_image_data()
data = kitten.get_data('RGB', kitten.width * 3)
```

data always returns a string, however it can be set to a ctypes array, stdlib array, list of byte data, string, or ctypes pointer. To set the image data use *set_data*, again specifying the format and pitch:

```
kitten.set_data('RGB', kitten.width * 3, data)
```

You can also create *ImageData* directly, by providing each of these attributes to the constructor. This is any easy way to load textures into OpenGL from other programs or libraries.

Performance concerns

pyglet can use several methods to transform pixel data from one format to another. It will always try to select the most efficient means. For example, when providing texture data to OpenGL, the following possibilities are examined in order:

1. Can the data be provided directly using a built-in OpenGL pixel format such as `GL_RGB` or `GL_RGBA`?
2. Is there an extension present that handles this pixel format?
3. Can the data be transformed with a single regular expression?
4. If none of the above are possible, the image will be split into separate scanlines and a regular expression replacement done on each; then the lines will be joined together again.

The following table shows which image formats can be used directly with steps 1 and 2 above, as long as the image rows are tightly packed (that is, the pitch is equal to the width times the number of components).

Format	Required extensions
"I "	
"L "	
"LA "	
"R "	
"G "	
"B "	
"A "	
"RGB "	
"RGBA "	
"ARGB "	GL_EXT_bgra and GL_APPLE_packed_pixels
"ABGR "	GL_EXT_abgr
"BGR "	GL_EXT_bgra
"BGRA "	GL_EXT_bgra

If the image data is not in one of these formats, a regular expression will be constructed to pull it into one. If the rows are not tightly packed, or if the image is ordered from top-to-bottom, the rows will be split before the regular expression is applied. Each of these may incur a performance penalty – you should avoid such formats for real-time texture updates if possible.

Image sequences and atlases

Sometimes a single image is used to hold several images. For example, a “sprite sheet” is an image that contains each animation frame required for a character sprite animation.

pyglet provides convenience classes for extracting the individual images from such a composite image as if it were a simple Python sequence. Discrete images can also be packed into one or more larger textures with texture bins and atlases.

Fig. 1.7: The AbstractImageSequence class hierarchy.

Image grids

An “image grid” is a single image which is divided into several smaller images by drawing an imaginary grid over it. The following image shows an image used for the explosion animation in the *Astraea* example.



Fig. 1.8: An image consisting of eight animation frames arranged in a grid.

This image has one row and eight columns. This is all the information you need to create an *ImageGrid* with:

```
explosion = pyglet.image.load('explosion.png')
explosion_seq = pyglet.image.ImageGrid(explosion, 1, 8)
```

The images within the grid can now be accessed as if they were their own images:

```
frame_1 = explosion_seq[0]
frame_2 = explosion_seq[1]
```

Images with more than one row can be accessed either as a single-dimensional sequence, or as a (row, column) tuple; as shown in the following diagram.

Fig. 1.9: An image grid with several rows and columns, and the slices that can be used to access it.

Image sequences can be sliced like any other sequence in Python. For example, the following obtains the first four frames in the animation:

```
start_frames = explosion_seq[:4]
```

For efficient rendering, you should use a *TextureGrid*. This uses a single texture for the grid, and each individual image returned from a slice will be a *TextureRegion*:

```
explosion_tex_seq = image.TextureGrid(explosion_seq)
```

Because *TextureGrid* is also a *Texture*, you can use it either as individual images or as the whole grid at once.

3D textures

TextureGrid is extremely efficient for drawing many sprites from a single texture. One problem you may encounter, however, is bleeding between adjacent images.

When OpenGL renders a texture to the screen, by default it obtains each pixel colour by interpolating nearby texels. You can disable this behaviour by switching to the `GL_NEAREST` interpolation mode, however you then lose the benefits of smooth scaling, distortion, rotation and sub-pixel positioning.

You can alleviate the problem by always leaving a 1-pixel clear border around each image frame. This will not solve the problem if you are using mipmapping, however. At this stage you will need a 3D texture.

You can create a 3D texture from any sequence of images, or from an *ImageGrid*. The images must all be of the same dimension, however they need not be powers of two (pyglet takes care of this by returning *TextureRegion* as with a regular *Texture*).

In the following example, the explosion texture from above is uploaded into a 3D texture:

```
explosion_3d = pyglet.image.Texture3D.create_for_image_grid(explosion_seq)
```

You could also have stored each image as a separate file and used *Texture3D.create_for_images* to create the 3D texture.

Once created, a 3D texture behaves like any other *ImageSequence*; slices return *TextureRegion* for an image plane within the texture. Unlike a *TextureGrid*, though, you cannot blit a *Texture3D* in its entirety.

Texture bins and atlases

Image grids are useful when the artist has good tools to construct the larger images of the appropriate format, and the contained images all have the same size. However it is often simpler to keep individual images as separate files on disk, and only combine them into larger textures at runtime for efficiency.

A *TextureAtlas* is initially an empty texture, but images of any size can be added to it at any time. The atlas takes care of tracking the “free” areas within the texture, and of placing images at appropriate locations within the texture to avoid overlap.

It’s possible for a *TextureAtlas* to run out of space for new images, so applications will need to either know the correct size of the texture to allocate initially, or maintain multiple atlases as each one fills up.

The *TextureBin* class provides a simple means to manage multiple atlases. The following example loads a list of images, then inserts those images into a texture bin. The resulting list is a list of *TextureRegion* images that map into the larger shared texture atlases:

```
images = [
    pyglet.image.load('img1.png'),
    pyglet.image.load('img2.png'),
    # ...
]

bin = pyglet.image.atlas.TextureBin()
images = [bin.add(image) for image in images]
```

The *pyglet.resource* module (see *Application resources*) uses texture bins internally to efficiently pack images automatically.

Animations

While image sequences and atlases provide storage for related images, they alone are not enough to describe a complete animation.

The *Animation* class manages a list of *AnimationFrame* objects, each of which references an image and a duration, in seconds. The storage of the images is up to the application developer: they can each be discrete, or packed into a texture atlas, or any other technique.

An animation can be loaded directly from a GIF 89a image file with *load_animation* (supported on Linux, Mac OS X and Windows) or constructed manually from a list of images or an image sequence using the class methods (in which case the timing information will also need to be provided). The *add_to_texture_bin* method provides a convenient way to pack the image frames into a texture bin for efficient access.

Individual frames can be accessed by the application for use with any kind of rendering, or the entire animation can be used directly with a *Sprite* (see next section).

The following example loads a GIF animation and packs the images in that animation into a texture bin. A sprite is used to display the animation in the window:

```
animation = pyglet.image.load_animation('animation.gif')
bin = pyglet.image.atlas.TextureBin()
animation.add_to_texture_bin(bin)
sprite = pyglet.sprite.Sprite(animation)

window = pyglet.window.Window()

@window.event
def on_draw():
    sprite.draw()

pyglet.app.run()
```

When animations are loaded with *pyglet.resource* (see *Application resources*) the frames are automatically packed into a texture bin.

This example program is located in *examples/programming_guide/animation.py*, along with a sample GIF animation file.

Buffer images

pyglet provides a basic representation of the framebuffer as components of the *AbstractImage* hierarchy. At this stage this representation is based off OpenGL 1.1, and there is no support for newer features such as framebuffer objects. Of course, this doesn't prevent you using framebuffer objects in your programs – *pyglet.gl* provides this functionality – just that they are not represented as *AbstractImage* types.

Fig. 1.10: The *BufferImage* hierarchy.

A framebuffer consists of

- One or more colour buffers, represented by *ColorBufferImage*
- An optional depth buffer, represented by *DepthBufferImage*
- An optional stencil buffer, with each bit represented by *BufferImageMask*
- Any number of auxilliary buffers, also represented by *ColorBufferImage*

You cannot create the buffer images directly; instead you must obtain instances via the *BufferManager*. Use *get_buffer_manager* to get this singleton:

```
buffers = image.get_buffer_manager()
```

Only the back-left color buffer can be obtained (i.e., the front buffer is inaccessible, and stereo contexts are not supported by the buffer manager):

```
color_buffer = buffers.get_color_buffer()
```

This buffer can be treated like any other image. For example, you could copy it to a texture, obtain its pixel data, save it to a file, and so on. Using the *texture* attribute is particularly useful, as it allows you to perform multipass rendering effects without needing a render-to-texture extension.

The depth buffer can be obtained similarly:

```
depth_buffer = buffers.get_depth_buffer()
```

When a depth buffer is converted to a texture, the class used will be a *DepthTexture*, suitable for use with shadow map techniques.

The auxilliary buffers and stencil bits are obtained by requesting one, which will then be marked as “in-use”. This permits multiple libraries and your application to work together without clashes in stencil bits or auxilliary buffer names. For example, to obtain a free stencil bit:

```
mask = buffers.get_buffer_mask()
```

The buffer manager maintains a weak reference to the buffer mask, so that when you release all references to it, it will be returned to the pool of available masks.

Similarly, a free auxilliary buffer is obtained:

```
aux_buffer = buffers.get_aux_buffer()
```

When using the stencil or auxilliary buffers, make sure you explicitly request these when creating the window. See *OpenGL configuration options* for details.

Displaying images

Images should be drawn into a window in the window's *on_draw* event handler. Usually a “sprite” should be created for each appearance of the image on-screen. Images can also be drawn directly without creating a sprite.

Sprites

A sprite is an instance of an image displayed in the window. Multiple sprites can share the same image; for example, hundreds of bullet sprites might share the same bullet image.

A sprite is constructed given an image or animation, and drawn with the *Sprite.draw* method:

```
sprite = pyglet.sprite.Sprite(image)

@window.event
def on_draw():
    window.clear()
    sprite.draw()
```

Sprites have properties for setting the position, rotation, scale, opacity, color tint and visibility of the displayed image. Sprites automatically handle displaying the most up-to-date frame of an animation. The following example uses a scheduled function to gradually move the sprite across the screen:

```
def update(dt):
    # Move 10 pixels per second
    sprite.x += dt * 10

# Call update 60 times a second
pyglet.clock.schedule_interval(update, 1/60.)
```

If you need to draw many sprites, use a *Batch* to draw them all at once. This is far more efficient than calling *draw* on each of them in a loop:

```
batch = pyglet.graphics.Batch()

sprites = [pyglet.sprite.Sprite(image, batch=batch),
           pyglet.sprite.Sprite(image, batch=batch),
           # ... ]

@window.event
def on_draw():
    window.clear()
    batch.draw()
```

When sprites are collected into a batch, no guarantee is made about the order in which they will be drawn. If you need to ensure some sprites are drawn before others (for example, landscape tiles might be drawn before character sprites, which might be drawn before some particle effect sprites), use two or more *OrderedGroup* objects to specify the draw order:

```
batch = pyglet.graphics.Batch()
background = pyglet.graphics.OrderedGroup(0)
foreground = pyglet.graphics.OrderedGroup(1)

sprites = [pyglet.sprite.Sprite(image, batch=batch, group=background),
           pyglet.sprite.Sprite(image, batch=batch, group=background),
           pyglet.sprite.Sprite(image, batch=batch, group=foreground),
           pyglet.sprite.Sprite(image, batch=batch, group=foreground),
           # ...]

@window.event
def on_draw():
    window.clear()
    batch.draw()
```

See the *Graphics* section for more details on batch and group rendering.

For best performance, try to collect all batch images into as few textures as possible; for example, by loading images with `pyglet.resource.image` (see *Application resources*) or with *Texture bins and atlases*).

Simple image blitting

A simple but less efficient way to draw an image directly into a window is with the *blit* method:

```
@window.event
def on_draw():
    window.clear()
    image.blit(x, y)
```

The *x* and *y* coordinates locate where to draw the anchor point of the image. For example, to center the image at (*x*, *y*):

```
kitten.anchor_x = kitten.width // 2
kitten.anchor_y = kitten.height // 2
kitten.blit(x, y)
```

You can also specify an optional *z* component to the *blit* method. This has no effect unless you have changed the default projection or enabled depth testing. In the following example, the second image is drawn *behind* the first, even though it is drawn after it:

```
from pyglet.gl import *
glEnable(GL_DEPTH_TEST)

kitten.blit(x, y, 0)
kitten.blit(x, y, -0.5)
```

The default pyglet projection has a depth range of (-1, 1) – images drawn with a *z* value outside this range will not be visible, regardless of whether depth testing is enabled or not.

Images with an alpha channel can be blended with the existing framebuffer. To do this you need to supply OpenGL with a blend equation. The following code fragment implements the most common form of alpha blending, however other techniques are also possible:

```
from pyglet.gl import *
glEnable(GL_BLEND)
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

You would only need to call the code above once during your program, before you draw any images (this is not necessary when using only sprites).

OpenGL imaging

This section assumes you are familiar with texture mapping in OpenGL (for example, chapter 9 of the [OpenGL Programming Guide](#)).

To create a texture from any *AbstractImage*, call *get_texture*:

```
kitten = image.load('kitten.jpg')
texture = kitten.get_texture()
```

Textures are automatically created and used by *ImageData* when blitted. It is useful to use textures directly when aiming for high performance or 3D applications.

The *Texture* class represents any texture object. The *target* attribute gives the texture target (for example, `GL_TEXTURE_2D`) and *id* the texture name. For example, to bind a texture:

```
glBindTexture(texture.target, texture.id)
```

Texture dimensions

Implementations of OpenGL prior to 2.0 require textures to have dimensions that are powers of two (i.e., 1, 2, 4, 8, 16, ...). Because of this restriction, pyglet will always create textures of these dimensions (there are several non-conformant post-2.0 implementations). This could have unexpected results for a user blitting a texture loaded from a file of non-standard dimensions. To remedy this, pyglet returns a *TextureRegion* of the larger texture corresponding to just the part of the texture covered by the original image.

A *TextureRegion* has an *owner* attribute that references the larger texture. The following session demonstrates this:

```
>>> rgba = image.load('tests/image/rgba.png')
>>> rgba
<ImageData 235x257>          # The image is 235x257
>>> rgba.get_texture()
<TextureRegion 235x257>     # The returned texture is a region
>>> rgba.get_texture().owner
<Texture 256x512>          # The owning texture has power-2 dimensions
>>>
```

A *TextureRegion* defines a *tex_coords* attribute that gives the texture coordinates to use for a quad mapping the whole image. *tex_coords* is a 4-tuple of 3-tuple of floats; i.e., each texture coordinate is given in 3 dimensions. The following code can be used to render a quad for a texture region:

```
texture = kitten.get_texture()
t = texture.tex_coords
w, h = texture.width, texture.height
array = (GLfloat * 32)(
    t[0][0], t[0][1], t[0][2], 1.,
    x,      y,      z,      1.,
    t[1][0], t[1][1], t[1][2], 1.,
    x + w,  y,      z,      1.,
    t[2][0], t[2][1], t[2][2], 1.,
    x + w,  y + h,  z,      1.,
    t[3][0], t[3][1], t[3][2], 1.,
    x,      y + h,  z,      1.)

glPushClientAttrib(GL_CLIENT_VERTEX_ARRAY_BIT)
glInterleavedArrays(GL_T4F_V4F, 0, array)
glDrawArrays(GL_QUADS, 0, 4)
glPopClientAttrib()
```

The *Texture.blit* method does this.

Use the *Texture.create* method to create either a texture region from a larger power-2 sized texture, or a texture with the exact dimensions using the `GL_texture_rectangle_ARB` extension.

Texture internal format

pyglet automatically selects an internal format for the texture based on the source image's *format* attribute. The following table describes how it is selected.

Format	Internal format
Any format with 3 components	GL_RGB
Any format with 2 components	GL_LUMINANCE_ALPHA
"A"	GL_ALPHA
"L"	GL_LUMINANCE
"I"	GL_INTENSITY
Any other format	GL_RGBA

Note that this table does not imply any mapping between format components and their OpenGL counterparts. For example, an image with format "RG" will use GL_LUMINANCE_ALPHA as its internal format; the luminance channel will be averaged from the red and green components, and the alpha channel will be empty (maximal).

Use the `Texture.create` class method to create a texture with a specific internal format.

Saving an image

Any image can be saved using the *save* method:

```
kitten.save('kitten.png')
```

or, specifying a file-like object:

```
kitten_stream = open('kitten.png', 'wb')
kitten.save('kitten.png', file=kitten_stream)
```

The following example shows how to grab a screenshot of your application window:

```
pyglet.image.get_buffer_manager().get_color_buffer().save('screenshot.png')
```

Note that images can only be saved in the PNG format unless PIL is installed.

Sound and video

pyglet can play many audio and video formats. Audio is played back with either OpenAL, DirectSound or Pulseaudio, permitting hardware-accelerated mixing and surround-sound 3D positioning. Video is played into OpenGL textures, and so can be easily be manipulated in real-time by applications and incorporated into 3D environments.

Decoding of compressed audio and video is provided by [AVbin](#), an optional component available for Linux, Windows and Mac OS X. AVbin needs to be installed separately.

If AVbin is not present, pyglet will fall back to reading uncompressed WAV files only. This may be sufficient for many applications that require only a small number of short sounds, in which case those applications need not distribute AVbin.

- *Audio drivers*
 - *DirectSound*
 - *OpenAL*
 - *Pulse*
 - *Linux Issues*
- *Supported media types*
- *Loading media*
- *Procedural Audio*
- *Simple audio playback*
- *Controlling playback*
- *Incorporating video*
- *Positional audio*

Audio drivers

pyglet can use OpenAL, DirectSound or Pulseaudio to play back audio. Only one of these drivers can be used in an application, and this must be selected before the `pyglet.media` module is loaded. The available drivers depend on your operating system:

Windows	Mac OS X	Linux
OpenAL ¹	OpenAL	OpenAL ¹
DirectSound		
		Pulseaudio

The audio driver can be set through the `audio` key of the `pyglet.options` dictionary. For example:

```
pyglet.options['audio'] = ('openal', 'silent')
```

This tells pyglet to use the OpenAL driver if it is available, and to ignore all audio output if it is not. The `audio` option can be a list of any of these strings, giving the preference order for each driver:

String	Audio driver
openal	OpenAL
directsound	DirectSound
pulse	Pulseaudio
silent	No audio output

You must set the `audio` option before importing `pyglet.media`. You can alternatively set it through an environment variable; see *Environment settings*.

The following sections describe the requirements and limitations of each audio driver.

DirectSound

DirectSound is available only on Windows, and is installed by default on Windows XP and later. pyglet uses only DirectX 7 features. On Windows Vista DirectSound does not support hardware audio mixing or surround sound.

OpenAL

OpenAL is included with Mac OS X. Windows users can download a generic driver from openal.org, or from their sound device's manufacturer. Linux users can use the reference implementation also provided by Creative. For

¹OpenAL is not installed by default on Windows, nor in many Linux distributions. It can be downloaded separately from your audio device manufacturer or openal.org

example, Ubuntu users can `apt-get install openal`. ALUT is not required. pyglet makes use of OpenAL 1.1 features if available, but will also work with OpenAL 1.0.

Due to a long-standing bug in the reference implementation of OpenAL, stereo audio is downmixed to mono on Linux. This does not affect Windows or Mac OS X users.

Pulse

Pulseaudio has become the standard Linux audio implementation over the past few years, and is installed by default with most modern Linux distributions.

Linux Issues

Linux users have the option of choosing between OpenAL and Pulse for audio output. Unfortunately OpenAL has severe limitations that are outside the scope of pyglet's control.

If your application can manage without stereo playback, you should use the OpenAL driver (assuming your users have it installed). You can do this with:

```
pyglet.options['audio'] = ('openal', 'pulse', 'silent')
```

If your application needs stereo playback, consider using the Pulse driver in preference to the OpenAL driver (this is the default).

Supported media types

If AVbin is not installed, only uncompressed RIFF/WAV files encoded with linear PCM can be read.

With AVbin, many common and less-common formats are supported. Due to the large number of combinations of audio and video codecs, options, and container formats, it is difficult to provide a complete yet useful list. Some of the supported audio formats are:

- AU
- MP2
- MP3
- OGG/Vorbis
- WAV
- WMA

Some of the supported video formats are:

- AVI
- DivX
- H.263
- H.264
- MPEG
- MPEG-2
- OGG/Theora
- Xvid

- WMV

For a complete list, see the AVbin sources. Otherwise, it is probably simpler to simply try playing back your target file with the `media_player.py` example.

New versions of AVbin as they are released may support additional formats, or fix errors in the current implementation. AVbin is completely future- and backward-compatible, so no change to pyglet is needed to use a newer version of AVbin – just install it in place of the old version.

Loading media

Audio and video files are loaded in the same way, using the `pyglet.media.load()` function, providing a file-name:

```
source = pyglet.media.load('explosion.wav')
```

If the media file is bundled with the application, consider using the resource module (see *Application resources*).

The result of loading a media file is a `Source` object. This object provides useful information about the type of media encoded in the file, and serves as an opaque object used for playing back the file (described in the next section).

The `load` function will raise a `MediaException` if the format is unknown. `IOError` may also be raised if the file could not be read from disk. Future versions of pyglet will also support reading from arbitrary file-like objects, however a valid filename must currently be given.

The length of the media file is given by the `duration` property, which returns the media's length in seconds.

Audio metadata is provided in the source's `audio_format` attribute, which is `None` for silent videos. This metadata is not generally useful to applications. See the `AudioFormat` class documentation for details.

Video metadata is provided in the source's `video_format` attribute, which is `None` for audio files. It is recommended that this attribute is checked before attempting play back a video file – if a movie file has a readable audio track but unknown video format it will appear as an audio file.

You can use the video metadata, described in a `VideoFormat` object, to set up display of the video before beginning playback. The attributes are as follows:

Attribute	Description
<code>width, height</code>	Width and height of the video image, in pixels.
<code>sample_aspect</code>	The aspect ratio of each video pixel.

You must take care to apply the sample aspect ratio to the video image size for display purposes. The following code determines the display size for a given video format:

```
def get_video_size(width, height, sample_aspect):
    if sample_aspect > 1.:
        return width * sample_aspect, height
    elif sample_aspect < 1.:
        return width, height / sample_aspect
    else:
        return width, height
```

Media files are not normally read entirely from disk; instead, they are streamed into the decoder, and then into the audio buffers and video memory only when needed. This reduces the startup time of loading a file and reduces the memory requirements of the application.

However, there are times when it is desirable to completely decode an audio file in memory first. For example, a sound that will be played many times (such as a bullet or explosion) should only be decoded once. You can instruct pyglet to completely decode an audio file into memory at load time:

```
explosion = pyglet.media.load('explosion.wav', streaming=False)
```

The resulting source is an instance of `StaticSource`, which provides the same interface as a streaming source. You can also construct a `StaticSource` directly from an already-loaded `Source`:

```
explosion = pyglet.media.StaticSource(pyglet.media.load('explosion.wav'))
```

Procedural Audio

In addition to loading audio files from disk, the `pyglet.media.procedural` module is available for creating various simple sounds. There are a variety of waveforms available:

- `pyglet.media.procedural.Sine`
- `pyglet.media.procedural.Saw`
- `pyglet.media.procedural.Square`
- `pyglet.media.procedural.FM`
- `pyglet.media.procedural.Silence`
- `pyglet.media.procedural.WhiteNoise`

At a minimum, you will need to specify the duration of the audio you wish to produce. You will also want to set the frequency (most waveforms will default to 440Hz). Some waveforms, such as the FM waveform, have additional parameters. More detail can be found in the API documentation for each.

Simple audio playback

Many applications, especially games, need to play sounds in their entirety without needing to keep track of them. For example, a sound needs to be played when the player's space ship explodes, but this sound never needs to have its volume adjusted, or be rewound, or interrupted.

pyglet provides a simple interface for this kind of use-case. Call the `play()` method of any `Source` to play it immediately and completely:

```
explosion = pyglet.media.load('explosion.wav', streaming=False)
explosion.play()
```

You can call `play` on any *Source*, not just *StaticSource*.

The return value of `Source.play` is a `Player`, which can either be discarded, or retained to maintain control over the sound's playback.

Controlling playback

You can implement many functions common to a media player using the `Player` class. Use of this class is also necessary for video playback. There are no parameters to its construction:

```
player = pyglet.media.Player()
```

A player will play any source that is “queued” on it. Any number of sources can be queued on a single player, but once queued, a source can never be dequeued (until it is removed automatically once complete). The main use of this queuing mechanism is to facilitate “gapless” transitions between playback of media files.

A `StreamingSource` can only ever be queued on one player, and only once on that player. `StaticSource` objects can be queued any number of times on any number of players. Recall that a `StaticSource` can be created by passing `streaming=False` to the `load` method.

In the following example, two sounds are queued onto a player:

```
player.queue(source1)
player.queue(source2)
```

Playback begins with the player's `play` method is called:

```
player.play()
```

Standard controls for controlling playback are provided by these methods:

Method	Description
<code>play</code>	Begin or resume playback of the current source.
<code>pause</code>	Pause playback of the current source.
<code>next_source</code>	Dequeue the current source and move to the next one immediately. <code>next</code> can also be used but it is deprecated because of incompatibilities with Python 3.
<code>seek</code>	Seek to a specific time within the current source.

Note that there is no `stop` method. If you do not need to resume playback, simply pause playback and discard the player and source objects. Using the `next_source()` method does not guarantee gapless playback.

There are several properties that describe the player's current state:

Property	Description
<code>time</code>	The current playback position within the current source, in seconds. This is read-only (but see the <code>seek</code> method).
<code>playing</code>	True if the player is currently playing, False if there are no sources queued or the player is paused. This is read-only (but see the <code>pause</code> and <code>play</code> methods).
<code>source</code>	A reference to the current source being played. This is read-only (but see the <code>queue</code> method).
<code>volume</code>	The audio level, expressed as a float from 0 (mute) to 1 (normal volume). This can be set at any time.

When a player reaches the end of the current source, by default it will move immediately to the next queued source. If there are no more sources, playback stops until another is queued. There are several other possible behaviours, which can be controlled on `SourceGroup` objects.

A `SourceGroup` contains multiple media sources with the same audio and video format. Behaviour on reaching the end of the current source can be controlled through the `loop` and `advance_after_eos` attributes.

You can change a `SourceGroup`'s `loop` and `advance_after_eos` at any time, but be aware that unless sufficient time is given for the future data to be decoded and buffered there may be a stutter or gap in playback. If set well in advance of the end of the source (say, several seconds), there will be no disruption.

Incorporating video

When a `Player` is playing back a source with video, use the `get_texture()` method to obtain the video frame image. This can be used to display the current video image synchronised with the audio track, for example:

```
@window.event
def on_draw():
    player.get_texture().blit(0, 0)
```

The texture is an instance of `pyglet.image.Texture`, with an internal format of either `GL_TEXTURE_2D` or `GL_TEXTURE_RECTANGLE_ARB`. While the texture will typically be created only once and subsequently updated

each frame, you should make no such assumption in your application – future versions of pyglet may use multiple texture objects.

Positional audio

pyglet uses OpenAL for audio playback, which includes many features for positioning sound within a 3D space. This is particularly effective with a surround-sound setup, but is also applicable to stereo systems.

A *Player* in pyglet has an associated position in 3D space – that is, it is equivalent to an OpenAL “source”. The properties for setting these parameters are described in more detail in the API documentation; see for example *Player.position* and *Player.pitch*.

The OpenAL “listener” object is provided by the audio driver. To obtain the listener for the current audio driver:

```
pyglet.media.get_audio_driver().get_listener()
```

This provides similar properties such as *Listener.position*, *Listener.forward_orientation* and *Listener.up_orientation* that describe the position of the user in 3D space.

Note that only mono sounds can be positioned. Stereo sounds will play back as normal, and only their volume and pitch properties will affect the sound.

Application resources

Previous sections in this guide have described how to load images, media and text documents using pyglet. Applications also usually have the need to load other data files: for example, level descriptions in a game, internationalised strings, and so on.

Programmers are often tempted to load, for example, an image required by their application with:

```
image = pyglet.image.load('logo.png')
```

This code assumes `logo.png` is in the current working directory. Unfortunately the working directory is not necessarily the same as the directory containing the application script files.

- Applications started from the command line can start from an arbitrary working directory.
- Applications bundled into an egg, Mac OS X package or Windows executable may have their resources inside a ZIP file.
- The application might need to change the working directory in order to work with the user’s files.

A common workaround for this is to construct a path relative to the script file instead of the working directory:

```
import os

script_dir = os.path.dirname(__file__)
path = os.path.join(script_dir, 'logo.png')
image = pyglet.image.load(path)
```

This, besides being tedious to write, still does not work for resources within ZIP files, and can be troublesome in projects that span multiple packages.

The *pyglet.resource* module solves this problem elegantly:

```
image = pyglet.resource.image('logo.png')
```

The following sections describe exactly how the resources are located, and how the behaviour can be customised.

Loading resources

Use the `pyglet.resource` module when files shipped with the application need to be loaded. For example, instead of writing:

```
data_file = open('file.txt')
```

use:

```
data_file = pyglet.resource.file('file.txt')
```

There are also convenience functions for loading media files for pyglet. The following table shows the equivalent resource functions for the standard file functions.

File function	Resource function	Type
<code>open</code>	<code>pyglet.resource.file</code>	File-like object
<code>pyglet.image.load</code>	<code>pyglet.resource.image</code>	<i>Texture</i> or <i>TextureRegion</i>
<code>pyglet.image.load</code>	<code>pyglet.resource.texture</code>	<i>Texture</i>
<code>pyglet.image.load_animation</code>	<code>pyglet.resource.animation</code>	<i>Animation</i>
<code>pyglet.media.load</code>	<code>pyglet.resource.media</code>	<i>Source</i>
<code>pyglet.text.load</code> <code>mimetype = text/plain</code>	<code>pyglet.resource.text</code>	<i>UnformattedDocument</i>
<code>pyglet.text.load</code> <code>mimetype = text/html</code>	<code>pyglet.resource.html</code>	<i>FormattedDocument</i>
<code>pyglet.text.load</code> <code>mimetype = text/vnd.pyglet-attributed</code>	<code>pyglet.resource.attributed</code>	<i>FormattedDocument</i>
<code>pyglet.font.add_file</code>	<code>pyglet.resource.add_font</code>	None

`pyglet.resource.texture` is for loading stand-alone textures, and would be required when using the texture for a 3D model.

`pyglet.resource.image` is optimised for loading sprite-like images that can have their texture coordinates adjusted. The resource module attempts to pack small images into larger textures (called an atlas) for efficient rendering (which is why the return type of this function can be *TextureRegion*).

Resource locations

Some resource files reference other files by name. For example, an HTML document can contain `` elements. In this case your application needs to locate `image.png` relative to the original HTML file.

Use `pyglet.resource.location` to get a *Location* object describing the location of an application resource. This location might be a file system directory or a directory within a ZIP file. The *Location* object can directly open files by name, so your application does not need to distinguish between these cases.

In the following example, a `thumbnails.txt` file is assumed to contain a list of image filenames (one per line), which are then loaded assuming the image files are located in the same directory as the `thumbnails.txt` file:

```
thumbnails_file = pyglet.resource.file('thumbnails.txt', 'rt')
thumbnails_location = pyglet.resource.location('thumbnails.txt')

for line in thumbnails_file:
    filename = line.strip()
    image_file = thumbnails_location.open(filename)
    image = pyglet.image.load(filename, file=image_file)
    # Do something with `image`...
```

This code correctly ignores other images with the same filename that might appear elsewhere on the resource path.

Specifying the resource path

By default, only the script home directory is searched (the directory containing the `__main__` module). You can set `pyglet.resource.path` to a list of locations to search in order. This list is indexed, so after modifying it you will need to call `pyglet.resource.reindex`.

Each item in the path list is either a path relative to the script home, or the name of a Python module preceded with an ampersand (`@`). For example, if you would like to package all your resources in a `res` directory:

```
pyglet.resource.path = ['res']
pyglet.resource.reindex()
```

Items on the path are not searched recursively, so if your resource directory itself has subdirectories, these need to be specified explicitly:

```
pyglet.resource.path = ['res', 'res/images', 'res/sounds', 'res/fonts']
pyglet.resource.reindex()
```

The entries in the resource path always use forward slash characters as path separators even when the operating systems using a different character.

Specifying module names makes it easy to group code with its resources. The following example uses the directory containing the hypothetical `gui.skins.default` for resources:

```
pyglet.resource.path = ['@gui.skins.default', '.']
pyglet.resource.reindex()
```

Multiple loaders

A *Loader* encapsulates a complete resource path and cache. This lets your application cleanly separate resource loading of different modules. Loaders are constructed for a given search path, and exposes the same methods as the global `pyglet.resource` module functions.

For example, if a module needs to load its own graphics but does not want to interfere with the rest of the application's resource loading, it would create its own *Loader* with a local search path:

```
loader = pyglet.resource.Loader(['@' + __name__])
image = loader.image('logo.png')
```

This is particularly suitable for “plugin” modules.

You can also use a *Loader* instance to load a set of resources relative to some user-specified document directory. The following example creates a loader for a directory specified on the command line:

```
import sys
home = sys.argv[1]
loader = pyglet.resource.Loader(script_home=[home])
```

This is the only way that absolute directories and resources not bundled with an application should be used with *pyglet.resource*.

Saving user preferences

Because Python applications can be distributed in several ways, including within ZIP files, it is usually not feasible to save user preferences, high score lists, and so on within the application directory (or worse, the working directory).

The *pyglet.resource.get_settings_path* function returns a directory suitable for writing arbitrary user-centric data. The directory used follows the operating system’s convention:

- `~/.config/AppName/` on Linux (depends on `XDG_CONFIG_HOME` environment variable).
- `$HOME\AppData\Local\Settings\AppName` on Windows
- `~/Library/Application Support/AppName` on Mac OS X

The returned directory name is not guaranteed to exist – it is the application’s responsibility to create it. The following example opens a high score list file for a game called “SuperGame” into the settings directory:

```
import os

dir = pyglet.resource.get_settings_path('SuperGame')
if not os.path.exists(dir):
    os.makedirs(dir)
filename = os.path.join(dir, 'highscores.txt')
file = open(filename, 'wt')
```

pyglet options

pyglet is a cross-platform games and multimedia package.

Detailed documentation is available at <http://www.pyglet.org>

options = {'debug_trace_args': False, 'search_local_libs': True, 'debug_trace_flush': True, 'debug_gl': True, 'debug_texture': True}

Global dict of pyglet options. To change an option from its default, you must import *pyglet* before any sub-packages. For example:

```
import pyglet
pyglet.options['debug_gl'] = False
```

The default options can be overridden from the OS environment. The corresponding environment variable for each option key is prefaced by `PYGLET_`. For example, in Bash you can set the `debug_gl` option with:

```
PYGLET_DEBUG_GL=True; export PYGLET_DEBUG_GL
```

For options requiring a tuple of values, separate each value with a comma.

The non-development options are:

audio A sequence of the names of audio modules to attempt to load, in order of preference. Valid driver names are:

- `directsound`, the Windows DirectSound audio module (Windows only)
- `pulse`, the PulseAudio module (Linux only)
- `openal`, the OpenAL audio module
- `silent`, no audio

debug_lib If True, prints the path of each dynamic library loaded.

debug_gl If True, all calls to OpenGL functions are checked afterwards for errors using `glGetError`. This will severely impact performance, but provides useful exceptions at the point of failure. By default, this option is enabled if `__debug__` is (i.e., if Python was not run with the `-O` option). It is disabled by default when pyglet is “frozen” within a py2exe or py2app library archive.

shadow_window By default, pyglet creates a hidden window with a GL context when `pyglet.gl` is imported. This allows resources to be loaded before the application window is created, and permits GL objects to be shared between windows even after they’ve been closed. You can disable the creation of the shadow window by setting this option to False.

Some OpenGL driver implementations may not support shared OpenGL contexts and may require disabling the shadow window (and all resources must be loaded after the window using them was created). Recommended for advanced developers only.

Since: pyglet 1.1

vsync If set, the `pyglet.window.Window.vsync` property is ignored, and this option overrides it (to either force vsync on or off). If unset, or set to None, the `pyglet.window.Window.vsync` property behaves as documented.

xsync If set (the default), pyglet will attempt to synchronise the drawing of double-buffered windows to the border updates of the X11 window manager. This improves the appearance of the window during re-size operations. This option only affects double-buffered windows on X11 servers supporting the Xsync extension with a window manager that implements the `_NET_WM_SYNC_REQUEST` protocol.

Since: pyglet 1.1

darwin_cocoa If True, the Cocoa-based pyglet implementation is used as opposed to the 32-bit Carbon implementation. When python is running in 64-bit mode on Mac OS X 10.6 or later, this option is set to True by default. Otherwise the Carbon implementation is preferred.

Since: pyglet 1.2

search_local_libs If False, pyglet won’t try to search for libraries in the script directory and its *lib* subdirectory. This is useful to load a local library instead of the system installed version. This option is set to True by default.

Since: pyglet 1.2

version = ‘1.2.2’

The release version of this pyglet installation.

Valid only if pyglet was installed from a source or binary distribution (i.e. not in a checked-out copy from SVN).

Use `setuptools` if you need to check for a specific release version, e.g.:

```
>>> import pyglet
>>> from pkg_resources import parse_version
>>> parse_version(pyglet.version) >= parse_version('1.1')
True
```

Debugging tools

pyglet includes a number of debug paths that can be enabled during or before application startup. These were primarily developed to aid in debugging pyglet itself, however some of them may also prove useful for understanding and debugging pyglet applications.

Each debug option is a key in the `pyglet.options` dictionary. Options can be set directly on the dictionary before any other modules are imported:

```
import pyglet
pyglet.options['debug_gl'] = False
```

They can also be set with environment variables before pyglet is imported. The corresponding environment variable for each option is the string `PYGLET_` prefixed to the uppercase option key. For example, the environment variable for `debug_gl` is `PYGLET_DEBUG_GL`. Boolean options are set or unset with 1 and 0 values.

A summary of the debug environment variables appears in the table below.

Option	Environment variable	Type
<code>debug_font</code>	<code>PYGLET_DEBUG_FONT</code>	bool
<code>debug_gl</code>	<code>PYGLET_DEBUG_GL</code>	bool
<code>debug_gl_trace</code>	<code>PYGLET_DEBUG_GL_TRACE</code>	bool
<code>debug_gl_trace_args</code>	<code>PYGLET_DEBUG_GL_TRACE_ARGS</code>	bool
<code>debug_graphics_batch</code>	<code>PYGLET_DEBUG_GRAPHICS_BATCH</code>	bool
<code>debug_lib</code>	<code>PYGLET_DEBUG_LIB</code>	bool
<code>debug_media</code>	<code>PYGLET_DEBUG_MEDIA</code>	bool
<code>debug_trace</code>	<code>PYGLET_DEBUG_TRACE</code>	bool
<code>debug_trace_args</code>	<code>PYGLET_DEBUG_TRACE_ARGS</code>	bool
<code>debug_trace_depth</code>	<code>PYGLET_DEBUG_TRACE_DEPTH</code>	int
<code>debug_win32</code>	<code>PYGLET_DEBUG_WIN32</code>	bool
<code>debug_x11</code>	<code>PYGLET_DEBUG_X11</code>	bool
<code>graphics_vbo</code>	<code>PYGLET_GRAPHICS_VBO</code>	bool

The `debug_media` and `debug_font` options are used to debug the `pyglet.media` and `pyglet.font` modules, respectively. Their behaviour is platform-dependent and useful only for pyglet developers.

The remaining debug options are detailed below.

Debugging OpenGL

The `graphics_vbo` option enables the use of vertex buffer objects in `pyglet.graphics` (instead, only vertex arrays). This is useful when debugging the `graphics` module as well as isolating code for determining if a video driver is faulty.

The `debug_graphics_batch` option causes all *Batch* objects to dump their rendering tree to standard output before drawing, after any change (so two drawings of the same tree will only dump once). This is useful to debug applications making use of *Group* and *Batch* rendering.

Error checking

The `debug_gl` option intercepts most OpenGL calls and calls `glGetError` afterwards (it only does this where such a call would be legal). If an error is reported, an exception is raised immediately.

This option is enabled by default unless the `-O` flag (optimisation) is given to Python, or the script is running from within a `py2exe` or `py2app` package.

Tracing

The `debug_gl_trace` option causes all OpenGL functions called to be dumped to standard out. When combined with `debug_gl_trace_args`, the arguments given to each function are also printed (they are abbreviated if necessary to avoid dumping large amounts of buffer data).

Tracing execution

The `debug_trace` option enables Python-wide function tracing. This causes every function call to be printed to standard out. Due to the large number of function calls required just to initialise pyglet, it is recommended to redirect standard output to a file when using this option.

The `debug_trace_args` option additionally prints the arguments to each function call.

When `debug_trace_depth` is greater than 1 the caller(s) of each function (and their arguments, if `debug_trace_args` is set) are also printed. Each caller is indented beneath the callee. The default depth is 1, specifying that no callers are printed.

Platform-specific debugging

The `debug_lib` option causes the path of each loaded library to be printed to standard out. This is performed by the undocumented `pyglet.lib` module, which on Linux and Mac OS X must sometimes follow complex procedures to find the correct library. On Windows not all libraries are loaded via this module, so they will not be printed (however, loading Windows DLLs is sufficiently simple that there is little need for this information).

Linux

X11 errors are caught by pyglet and suppressed, as there are plenty of X servers in the wild that generate errors that can be safely ignored. The `debug_x11` option causes these errors to be dumped to standard out, along with a traceback of the Python stack (this may or may not correspond to the error, depending on whether or not it was reported asynchronously).

Windows

The `debug_win32` option causes all library calls into `user32.dll`, `kernel32.dll` and `gdi32.dll` to be intercepted. Before each library call `SetLastError(0)` is called, and afterwards `GetLastError()` is called. Any errors discovered are written to a file named `debug_win32.log`. Note that an error is only valid if the function called returned an error code, but the interception function does not check this.

Advanced topics

- *Environment settings*

Environment settings

Options in the `pyglet.options` dictionary can have defaults set through the operating system's environment variable. The following table shows which environment variable is used for each option:

Environment variable	<i>pyglet.options</i> key	Type	Default value
PYGLET_AUDIO	audio	List of strings	directsound, opeanl, alsa, silent
PYGLET_DEBUG_GL	debug_gl	Boolean	1 ¹

¹ Defaults to 1 unless Python is run with `-O` or from a frozen executable.

Appendix: Migrating to pyglet 1.1

pyglet 1.1 introduces new features for rendering high performance graphics and text, is more convenient to use, and integrates better with the operating system. Some of the existing interfaces have also been redesigned slightly to conform with standard Python practice or to fix design flaws.

- *Compatibility and deprecation*
- *Deprecated methods*
- *New features replacing standard practice*
 - *Importing pyglet*
 - *Application event loop*
 - *Loading resources*
- *New graphics features*
- *New text features*
- *Other new features*

Compatibility and deprecation

pyglet 1.1 is backward compatible with pyglet 1.0. Any application that uses only public and documented methods of pyglet 1.0 will continue to work unchanged in pyglet 1.1. If you encounter an issue where this is not the case, please consider it a bug in pyglet and file an issue report.

Some methods have been marked *deprecated* in pyglet 1.1. These methods continue to work, but have been superseded by newer methods that are either more efficient or have a better design. The API reference has a complete list of deprecated methods; the main changes are described in the next section.

- Continue to use deprecated methods if your application needs to work with pyglet 1.0 as well as pyglet 1.1.
- New applications should not use deprecated methods.

Deprecated methods will continue to be supported in all minor revisions of pyglet 1.x. A pyglet 2.0 release will no longer support these methods.

Deprecated methods

The following minor changes have been made for design or efficiency reasons. Applications which no longer need to support pyglet 1.0 should make the appropriate changes to ensure the deprecated methods are not called.

The `dispatch_events` method on *Player* and the equivalent function on the *pyglet.media* module should no longer be called. In pyglet 1.1, media objects schedule an update function on *pyglet.clock* at an appropriate interval. New applications using media are required to call *pyglet.clock.tick* periodically.

The *AbstractImage* properties `texture`, `image_data`, and so on have been replaced with equivalent methods `get_texture`, `get_image_data`, etc.

The *ImageData* properties `data`, `format` and `pitch`, which together were used to extract pixel data from an image, have been replaced with a single function `get_data`. The `format` and `pitch` properties should now be used only to determine the current format and pitch of the image.

The `get_current_context` function has been replaced with a global variable, `current_context`, for efficiency.

New features replacing standard practice

pyglet 1.1 introduces new features that make it easier to program with, so the standard practice as followed in many of the pyglet example programs has changed.

Importing pyglet

In pyglet 1.0, it was necessary to explicitly import each submodule required by the application; for example:

```
from pyglet import font
from pyglet import image
from pyglet import window
```

pyglet now lazily loads submodules on demand, so an application can get away with importing just *pyglet*. This is especially handy for modules that are typically only used once in an application, and frees up the names *font*, *image*, *window* and so on for the application developer. For example:

```
window = pyglet.window.Window()
```

Application event loop

Every application using pyglet 1.0 provides its own event loop, such as:

```
while not window.has_exit:
    dt = clock.tick()
    update(dt)

    window.dispatch_events()
    window.clear()
    draw()
    window.flip()
```

Besides being somewhat repetitious to type, this type of event loop is difficult to extend with more windows, and exhausts all available system resources, even if the application is not doing anything.

The new *pyglet.app* module provides an application event loop that is less demanding of the CPU yet more responsive to user events. A complete application that opens an empty window can be written with:

```
window = pyglet.window.Window()

@window.event
def on_draw():
    window.clear()

pyglet.app.run()
```

Note the new *on_draw* event, which makes it easy to specify different drawing functions for each window. The *pyglet.app* event loop takes care of dispatching events, ticking the clock, calling the draw function and flipping the window buffer.

Update functions can be scheduled on the clock. To have an update function be called as often as possible, use *clock.schedule* (this effectively degenerates into the older *dispatch_events* practice of thrashing the CPU):

```
def update(dt):
    pass
clock.schedule(update)
```


Usually applications can update at a less frequent interval. For example, a game that is designed to run at 60Hz can use `clock.schedule_interval`:

```
def update(dt):
    pass
clock.schedule_interval(update, 1/60.0)
```

This also removes the need for `clock.set_fps_limit`.

Besides the advantages already listed, windows managed by the event loop will not block while being resized or moved; and the menu bar on OS X can be interacted with without blocking the application.

It is highly recommended that all applications use the event loop. The loop can be extended if you need to add additional hooks or integrate with another package. Applications continuing to use `Window.dispatch_events` gain no advantage, but suffer from poorer response, increased CPU usage and artifacts during window resizing and moving.

See *The application event loop* for more details.

Loading resources

Locating resources such as images, sound and video files, data files and fonts is difficult to do correctly across all platforms, considering the effects of a changing working directory and various distribution packages such as `setuptools`, `py2exe` and `py2app`.

The new `pyglet.resource` module implements the correct logic for all these cases, making it simple to load resources that belong to a specific module or the application as a whole. A resource path can be set that is indexed once, and can include filesystem directories, Python module paths and ZIP files.

For example, suppose your application ships with a `logo.png` that needs to be loaded on startup. In pyglet 1.0 you might have written:

```
import os.path
from pyglet import image

script_dir = os.path.dirname(__file__)
logo_filename = os.path.join(script_dir, 'logo.png')
logo = image.load(logo_filename)
```

In pyglet 1.1, you can write:

```
logo = pyglet.resource.image('logo.png')
```

And will actually work in more scenarios (such as within a `setuptools` egg file, `py2exe` and `py2app`).

The resource module efficiently packs multiple small images into larger textures, so there is less need for artists to create sprite sheets themselves for efficient rendering. Images and textures are also cached automatically.

See *Application resources* for more details.

New graphics features

The `pyglet.graphics` module is a low-level abstraction of OpenGL vertex arrays and buffer objects. It is intended for use by developers who are already very familiar with OpenGL and are after the best performance possible. pyglet uses this module internally to implement its new sprite module and the new text rendering module. The *Graphics* chapter describes this module in detail.

The `pyglet.sprite` module provide a fast, easy way to display 2D graphics on screen. Sprites can be moved, rotated, scaled and made translucent. Using the *batch* features of the new graphics API, multiple sprites can be drawn in one go very quickly. See *Sprites* for details.

The `pyglet.image.load_animation` function can load animated GIF images. These are returned as an *Animation*, which exposes the individual image frames and timings. Animations can also be played directly on a sprite in place of an image. The *Animations* chapter describes how to use them.

The `pyglet.image.atlas` module packs multiple images into larger textures for efficient rendering. The `pyglet.resource` module uses this module for small images automatically, but you can use it directly even if you're not making use of `pyglet.resource`. See *Texture bins and atlases* for details.

Images now have `anchor_x` and `anchor_y` attributes, which specify a point from which the image should be drawn. The `sprite` module also uses the anchor point as the center of rotation.

Textures have a `get_transform` method for retrieving a *TextureRegion* that refers to the same texture data in video memory, but with optional horizontal or vertical flipping, or 90-degree rotation.

New text features

The `pyglet.text` module can render formatted text efficiently. A new class *Label* supercedes the old `pyglet.font.Text` class (which is now actually implemented in terms of *Label*). The “Hello, World” application can now be written:

```
window = pyglet.window.Window()
label = pyglet.text.Label('Hello, world',
                          font_name='Times New Roman',
                          font_size=36,
                          x=window.width//2, y=window.height//2,
                          halign='center', valign='center')

@window.event
def on_draw():
    window.clear()
    label.draw()

pyglet.app.run()
```

You can also display multiple fonts and styles within one label, with *HTMLLabel*:

```
label = pyglet.text.HTMLLabel('<b>Hello</b>, <font color=red>world!</font>')
```

More advanced uses of the new text module permit applications to efficiently display large, scrolling, formatted documents (for example, HTML files with embedded images), and to allow the user to interactively edit text as in a WYSIWYG text editor.

Other new features

EventDispatcher now has a `remove_handlers` method which provides finer control over the event stack than `pop_handlers`.

The `@event` decorator has been fixed so that it no longer overrides existing event handlers on the object, which fixes the common problem of handling the `on_resize` event. For example, the following now works without any surprises (in `pyglet 1.0` this would override the default handler, which sets up a default, necessary viewport and projection):

```
@window.event
def on_resize(width, height):
    pass
```

A variant of `clock.schedule_interval`, `clock.schedule_interval_soft` has been added. This is for functions that need to be called periodically at a given interval, but do not need to schedule the period immediately. Soft interval scheduling is used by the `pyglet.media` module to distribute the work of decoding video and audio data over time, rather than stalling

the CPU periodically. Games could use soft interval scheduling to spread the regular computational requirements of multiple agents out over time.

In pyglet 1.0, *font.load* attempted to match the font resolution (DPI) with the operating system's typical behaviour. For example, on Linux and Mac OS X the default DPI was typically set at 72, and on Windows at 96. While this would be useful for writing a word processor, it adds a burden on the application developer to ensure their fonts work at arbitrary resolutions. In pyglet 1.1 the default DPI is set at 96 across all platforms. It can still be overridden explicitly by the application if desired.

Video sources in *pyglet.media* can now be stepped through frame-by-frame: individual image frames can be extracted without needing to play back the video in realtime.

For a complete list of new features and bug fixes, see the `CHANGELOG` distributed with the source distribution.

API Reference

pyglet

pyglet is a cross-platform games and multimedia package.

Detailed documentation is available at <http://www.pyglet.org>

Modules

<i>app</i>	Application-wide functionality.
<i>canvas</i>	Display and screen management.
<i>clock</i>	Precise framerate calculation, scheduling and framerate limiting.
<i>debug</i>	
<i>event</i>	Event dispatch framework.
<i>font</i>	Load fonts and render text.
<i>gl</i>	OpenGL and GLU interface.
<i>graphics</i>	Low-level graphics rendering.
<i>image</i>	Image load, capture and high-level texture functions.
<i>info</i>	Get environment information useful for debugging.
<i>input</i>	Joystick, tablet and USB HID device support.
<i>media</i>	Audio and video playback.
<i>resource</i>	Load application resources from a known path.
<i>sprite</i>	Display positioned, scaled and rotated images.
<i>text</i>	Text formatting, layout and display.
<i>window</i>	Windowing and user-interface events.

`pyglet.app`

Application-wide functionality.

Applications

Most applications need only call `run()` after creating one or more windows to begin processing events. For example, a simple application consisting of one window is:

```
import pyglet

win = pyglet.window.Window()
pyglet.app.run()
```

Events To handle events on the main event loop, instantiate it manually. The following example exits the application as soon as any window is closed (the default policy is to wait until all windows are closed):

```
event_loop = pyglet.app.EventLoop()

@event_loop.event
def on_window_close(window):
    event_loop.exit()
```

Note: Since pyglet 1.1

event_loop is the global event loop. Applications can replace this with their own subclass of `EventLoop` before calling `EventLoop.run()`.

platform_event_loop is the platform-dependent event loop. Applications must not subclass or replace this `PlatformEventLoop` object.

Modules

base

`pyglet.app.base`

<i>EventLoop</i>	The main run loop of the application.
<i>PlatformEventLoop</i>	Abstract class, implementation depends on platform.

Classes



EventLoop Class

class `EventLoop`

The main run loop of the application.

Calling *run* begins the application event loop, which processes operating system events, calls *pyglet.clock.tick* to call scheduled functions and calls *pyglet.window.Window.on_draw* and *pyglet.window.Window.flip* to update window contents.

Applications can subclass *EventLoop* and override certain methods to integrate another framework's run loop, or to customise processing in some other way. You should not in general override *run*, as this method contains platform-specific code that ensures the application remains responsive to the user while keeping CPU usage to a minimum.

Methods:

Attributes:

<i>event_types</i>	
<i>has_exit</i>	Flag indicating if the event loop will exit in the next iteration.

Attributes

`EventLoop.event_types = ['on_window_close', 'on_enter', 'on_exit']`

`EventLoop.has_exit`

Flag indicating if the event loop will exit in the next iteration. When set, all waiting threads are interrupted (see *sleep*).

Thread-safe since pyglet 1.2.

See *exit*

Type bool

Inherited members

Methods

`EventLoop.register_event_type(name)`

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters *name* (*str*) – Name of the event to register.

pyglet.app.base.PlatformEventLoop

PlatformEventLoop Class

class `PlatformEventLoop`

Abstract class, implementation depends on platform.

Note: Since pyglet 1.2

Variables

app = <pyglet._ModuleProxy object>

clock = <pyglet._ModuleProxy object>

compat_platform = 'linux'

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

division = _Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)

event = <module 'pyglet.event' from '/home/docs/checkouts/readthedocs.org/user_builds/pyglet/checkouts/latest/pyglet/event.py'>

Event dispatch framework.

All objects that produce events in pyglet implement *EventDispatcher*, providing a consistent interface for registering and manipulating event handlers. A commonly used event dispatcher is *pyglet.window.Window*.

For each event dispatcher there is a set of events that it dispatches; these correspond with the type of event handlers you can attach. Event types are identified by their name, for example, "on_resize". If you are creating a new class which implements *EventDispatcher*, you must call *EventDispatcher.register_event_type* for each event type.

An event handler is simply a function or method. You can attach an event handler by setting the appropriate function on the instance:

```
def on_resize(width, height):
    # ...
dispatcher.on_resize = on_resize
```

There is also a convenience decorator that reduces typing:

```
@dispatcher.event
def on_resize(width, height):
    # ...
```

You may prefer to subclass and override the event handlers instead:

```
class MyDispatcher(DispatcherClass):
    def on_resize(self, width, height):
        # ...
```

When attaching an event handler to a dispatcher using the above methods, it replaces any existing handler (causing the original handler to no longer be called). Each dispatcher maintains a stack of event handlers, allowing you to insert an event handler "above" the existing one rather than replacing it.

There are two main use cases for "pushing" event handlers:

- Temporarily intercepting the events coming from the dispatcher by pushing a custom set of handlers onto the dispatcher, then later "popping" them all off at once.
- Creating "chains" of event handlers, where the event propagates from the top-most (most recently added) handler to the bottom, until a handler takes care of it.

Use *EventDispatcher.push_handlers* to create a new level in the stack and attach handlers to it. You can push several handlers at once:

```
dispatcher.push_handlers(on_resize, on_key_press)
```

If your function handlers have different names to the events they handle, use keyword arguments:

```
dispatcher.push_handlers(on_resize=my_resize,
                        on_key_press=my_key_press)
```

After an event handler has processed an event, it is passed on to the next-lowest event handler, unless the handler returns *EVENT_HANDLED*, which prevents further propagation.

To remove all handlers on the top stack level, use *EventDispatcher.pop_handlers*.

Note that any handlers pushed onto the stack have precedence over the handlers set directly on the instance (for example, using the methods described in the previous section), regardless of when they were set. For example, handler `foo` is called before handler `bar` in the following example:

```
dispatcher.push_handlers(on_resize=foo)
dispatcher.on_resize = bar
```

pyglet uses a single-threaded model for all application code. Event handlers are only ever invoked as a result of calling *EventDispatcher.dispatch_events*.

It is up to the specific event dispatcher to queue relevant events until they can be dispatched, at which point the handlers are called in the order the events were originally generated.

This implies that your application runs with a main loop that continuously updates the application state and checks for new events:

```
while True:
    dispatcher.dispatch_events()
    # ... additional per-frame processing
```

Not all event dispatchers require the call to *dispatch_events*; check with the particular class documentation.

```
print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)
```

Defined

Notes

- platform
- queue
- standard_library
- sys
- threading

Classes

WeakSet Set of objects, referenced weakly.

pyglet.app.WeakSet

***WeakSet* Class**

class WeakSet

Set of objects, referenced weakly.

Adding an object to this set does not prevent it from being garbage collected. Upon being garbage collected, the object is automatically removed from the set.

Exceptions

AppException

pyglet.app.AppException

AppException

Exception defined in *pyglet.app*

exception AppException

Functions

<i>exit()</i>	Exit the application event loop.
<i>run()</i>	Begin processing events, scheduled functions and window updates.

***exit* Function** Defined in *pyglet.app*

exit()

Exit the application event loop.

Causes the application event loop to finish, if an event loop is currently running. The application may not necessarily exit (for example, there may be additional code following the *run* invocation).

This is a convenience function, equivalent to:

event_loop.exit()

***run* Function** Defined in *pyglet.app*

run()

Begin processing events, scheduled functions and window updates.

This is a convenience function, equivalent to:

```
pyglet.app.event_loop.run()
```

Variables

compat_platform = 'linux'

`str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str`

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of `object.__str__()` (if defined) or `repr(object)`. encoding defaults to `sys.getdefaultencoding()`. errors defaults to 'strict'.

displays = <pyglet.app.WeakSet object>

Set of all open displays. Instances of `pyglet.canvas.Display` are automatically added to this set upon construction. The set uses weak references, so displays are removed from the set when they are no longer referenced.

Warning: Deprecated. Use `pyglet.canvas.get_display()`.

Type *WeakSet*

event_loop = <pyglet.app.base.EventLoop object>

The main run loop of the application.

Calling *run* begins the application event loop, which processes operating system events, calls *pyglet.clock.tick* to call scheduled functions and calls *pyglet.window.Window.on_draw* and *pyglet.window.Window.flip* to update window contents.

Applications can subclass *EventLoop* and override certain methods to integrate another framework's run loop, or to customise processing in some other way. You should not in general override *run*, as this method contains platform-specific code that ensures the application remains responsive to the user while keeping CPU usage to a minimum.

platform_event_loop = <pyglet.app.base.PlatformEventLoop object>

Abstract class, implementation depends on platform.

Note: Since pyglet 1.2

windows = <pyglet.app.WeakSet object>

Set of all open windows (including invisible windows). Instances of *pyglet.window.Window* are automatically added to this set upon construction. The set uses weak references, so windows are removed from the set when they are no longer referenced or are closed explicitly.

Notes

Defined

- `sys`
- `weakref`

pyglet.canvas

Display and screen management.

Rendering is performed on a `Canvas`, which conceptually could be an off-screen buffer, the content area of a `pyglet.window.Window`, or an entire screen. Currently, canvases can only be created with windows (though windows can be set fullscreen).

Windows and canvases must belong to a `Display`. On Windows and Mac OS X there is only one display, which can be obtained with `get_display()`. Linux supports multiple displays, corresponding to discrete X11 display connections and screens. `get_display()` on Linux returns the default display and screen 0 (`localhost:0.0`); if a particular screen or display is required then `Display` can be instantiated directly.

Within a display one or more screens are attached. A `Screen` often corresponds to a physical attached monitor, however a monitor or projector set up to clone another screen will not be listed. Use `Display.get_screens()` to get a list of the attached screens; these can then be queried for their sizes and virtual positions on the desktop.

The size of a screen is determined by its current mode, which can be changed by the application; see the documentation for `Screen`.

Note: Since pyglet 1.2

Modules

base

pyglet.canvas.base

<i>Canvas</i>	Abstract drawing area.
<i>Display</i>	A display device supporting one or more screens.
<i>Screen</i>	A virtual monitor that supports fullscreen windows.
<i>ScreenMode</i>	Screen resolution and display settings.

Classes

pyglet.canvas.base.Canvas

Canvas Class

class Canvas (*display*)

Abstract drawing area.

Canvases are used internally by pyglet to represent drawing areas – either within a window or full-screen.

Note: Since pyglet 1.2

Instance Attributes

`Canvas.display`

Display this canvas was created on.

`pyglet.canvas.base.Display`

Display Class

class Display (*name=None, x_screen=None*)

A display device supporting one or more screens.

See also:

Programming Guide - *Displays*

Note: Since pyglet 1.2

Attributes:

<i>name</i>	Name of this display, if applicable.
<i>x_screen</i>	The X11 screen number of this display, if applicable.

Attributes

`Display.name = None`

Name of this display, if applicable.

Type str

`Display.x_screen = None`

The X11 screen number of this display, if applicable.

Type int

`pyglet.canvas.base.Screen`

Screen Class

class Screen (*display, x, y, width, height*)

A virtual monitor that supports fullscreen windows.

Screens typically map onto a physical display such as a monitor, television or projector. Selecting a screen for a window has no effect unless the window is made fullscreen, in which case the window will fill only that particular virtual screen.

The *width* and *height* attributes of a screen give the current resolution of the screen. The *x* and *y* attributes give the global location of the top-left corner of the screen. This is useful for determining if screens are arranged above or next to one another.

Use `get_screens()` or `get_default_screen()` to obtain an instance of this class.

See also:

Programming Guide - *Screens*

Instance Attributes

Screen.display

Display this screen belongs to.

Screen.height

Height of the screen, in pixels.

Screen.width

Width of the screen, in pixels.

Screen.x

Left edge of the screen on the virtual desktop.

Screen.y

Top edge of the screen on the virtual desktop.

`pyglet.canvas.base.ScreenMode`

ScreenMode Class

class ScreenMode (*screen*)

Screen resolution and display settings.

Applications should not construct *ScreenMode* instances themselves; see `Screen.get_modes()`.

The *depth* and *rate* variables may be `None` if the operating system does not provide relevant data.

Note: Since pyglet 1.2

Attributes:

<i>depth</i>	Pixel color depth, in bits per pixel.
<i>height</i>	Height of screen, in pixels.
<i>rate</i>	Screen refresh rate in Hz.
<i>width</i>	Width of screen, in pixels.

Attributes

`ScreenMode.depth = None`
Pixel color depth, in bits per pixel.

Type `int`
`ScreenMode.height = None`
Height of screen, in pixels.

Type `int`
`ScreenMode.rate = None`
Screen refresh rate in Hz.

Type `int`
`ScreenMode.width = None`
Width of screen, in pixels.
Type `int`

Variables

`gl = <module 'pyglet.gl' from '/home/docs/checkouts/readthedocs.org/user_builds/pyglet/checkouts/latest/pyglet/gl/__init__.py'>`
OpenGL and GLU interface.

This package imports all OpenGL, GLU and registered OpenGL extension functions. Functions have identical signatures to their C counterparts. For example:

```
from pyglet.gl import *

# [...omitted: set up a GL context and framebuffer]
glBegin(GL_QUADS)
glVertex3f(0, 0, 0)
glVertex3f(0.1, 0.2, 0.3)
glVertex3f(0.1, 0.2, 0.3)
glEnd()
```

OpenGL is documented in full at the [OpenGL Reference Pages](#).

The [OpenGL Programming Guide](#) is a popular reference manual organised by topic. The free online version documents only OpenGL 1.1. [Later editions](#) cover more recent versions of the API and can be purchased from a book store.

The following subpackages are imported into this “mega” package already (and so are available by importing `pyglet.gl`):

`pyglet.gl.gl` OpenGL

`pyglet.gl.glu` GLU

`pyglet.gl.gl.glext_arb` ARB registered OpenGL extension functions

These subpackages are also available, but are not imported into this namespace by default:

`pyglet.gl.glext_nv` nVidia OpenGL extension functions

`pyglet.gl.agl` AGL (Mac OS X OpenGL context functions)

`pyglet.gl.glx` GLX (Linux OpenGL context functions)

`pyglet.gl.glxext_arb` ARB registered GLX extension functions

pyglet.gl.glxext_nv nvidia GLX extension functions

pyglet.gl.wgl WGL (Windows OpenGL context functions)

pyglet.gl.wglxext_arb ARB registered WGL extension functions

pyglet.gl.wglxext_nv nvidia WGL extension functions

The information modules are provided for convenience, and are documented below.

window = <module 'pyglet.window' from '/home/docs/checkouts/readthedocs.org/user_builds/pyglet/checkouts/latest/pyglet/wi

Windowing and user-interface events.

This module allows applications to create and display windows with an OpenGL context. Windows can be created with a variety of border styles or set fullscreen.

You can register event handlers for keyboard, mouse and window events. For games and kiosks you can also restrict the input to your windows, for example disabling users from switching away from the application with certain key combinations or capturing and hiding the mouse.

Call the Window constructor to create a new window:

```
from pyglet.window import Window
win = Window(width=640, height=480)
```

Attach your own event handlers:

```
@win.event
def on_key_press(symbol, modifiers):
    # ... handle this event ...
```

Place drawing code for the window within the *Window.on_draw* event handler:

```
@win.event
def on_draw():
    # ... drawing code ...
```

Call *pyglet.app.run* to enter the main event loop (by default, this returns when all open windows are closed):

```
from pyglet import app
app.run()
```

Use *Window.set_exclusive_mouse* to hide the mouse cursor and receive relative mouse movement events. Specify *fullscreen=True* as a keyword argument to the *Window* constructor to render to the entire screen rather than opening a window:

```
win = Window(fullscreen=True)
win.set_exclusive_mouse()
```

By default, fullscreen windows are opened on the primary display (typically set by the user in their operating system settings). You can retrieve a list of attached screens and select one manually if you prefer. This is useful for opening a fullscreen window on each screen:

```
display = window.get_platform().get_default_display()
screens = display.get_screens()
windows = []
for screen in screens:
    windows.append(window.Window(fullscreen=True, screen=screen))
```

Specifying a screen has no effect if the window is not fullscreen.

Each window has its own context which is created when the window is created. You can specify the properties of the context before it is created by creating a “template” configuration:


```

from pyglet import gl
# Create template config
config = gl.Config()
config.stencil_size = 8
config.aux_buffers = 4
# Create a window using this config
win = window.Window(config=config)

```

To determine if a given configuration is supported, query the screen (see above, “Working with multiple screens”):

```

configs = screen.get_matching_configs(config)
if not configs:
    # ... config is not supported
else:
    win = window.Window(config=configs[0])

```

Defined

Notes

- app

Classes

Functions

[get_display\(\)](#) Get the default display device.

[get_display](#) Function Defined in `pyglet.canvas`

`get_display()`

Get the default display device.

If there is already a `Display` connection, that display will be returned. Otherwise, a default `Display` is created and returned. If multiple display connections are active, an arbitrary one is returned.

Note: Since pyglet 1.2

Return type `Display`

Notes

Defined

- sys

pyglet.clock

Precise framerate calculation, scheduling and framerate limiting.

Measuring time

The *tick* and *get_fps* functions can be used in conjunction to fulfil most games' basic requirements:

```
from pyglet import clock
while True:
    dt = clock.tick()
    # ... update and render ...
    print 'FPS is %f' % clock.get_fps()
```

The *dt* value returned gives the number of seconds (as a float) since the last “tick”.

The *get_fps* function averages the framerate over a sliding window of approximately 1 second. (You can calculate the instantaneous framerate by taking the reciprocal of *dt*).

Always remember to *tick* the clock!

Limiting frame-rate

The framerate can be limited:

```
clock.set_fps_limit(60)
```

This causes *clock* to sleep during each *tick* in an attempt to keep the number of ticks (frames) per second below 60.

The implementation uses platform-dependent high-resolution sleep functions to achieve better accuracy with busy-waiting than would be possible using just the *time* module.

Scheduling

You can schedule a function to be called every time the clock is ticked:

```
def callback(dt):
    print '%f seconds since last callback' % dt

clock.schedule(callback)
```

The *schedule_interval* method causes a function to be called every “n” seconds:

```
clock.schedule_interval(callback, .5)    # called twice a second
```

The *schedule_once* method causes a function to be called once “n” seconds in the future:

```
clock.schedule_once(callback, 5)        # called in 5 seconds
```

All of the *schedule* methods will pass on any additional args or keyword args you specify to the callback function:

```
def animate(dt, velocity, sprite):
    sprite.position += dt * velocity

clock.schedule(animate, velocity=5.0, sprite=alien)
```

You can cancel a function scheduled with any of these methods using *unschedule*:

```
clock.unschedule(animate)
```

Displaying FPS

The `ClockDisplay` class provides a simple FPS counter. You should create an instance of `ClockDisplay` once during the application's start up:

```
fps_display = clock.ClockDisplay()
```

Call `draw` on the `ClockDisplay` object for each frame:

```
fps_display.draw()
```

There are several options to change the font, color and text displayed within the `__init__` method.

Using multiple clocks

The clock functions are all relayed to an instance of `Clock` which is initialised with the module. You can get this instance to use directly:

```
clk = clock.get_default()
```

You can also replace the default clock with your own:

```
myclk = clock.Clock() clock.set_default(myclk)
```

Each clock maintains its own set of scheduled functions and FPS limiting/measurement. Each clock must be “ticked” separately.

Multiple and derived clocks potentially allow you to separate “game-time” and “wall-time”, or to synchronise your clock to an audio or video stream instead of the system clock.

Classes

<i><code>Clock</code></i>	Class for calculating and limiting framerate, and for calling scheduled functions.
<i><code>ClockDisplay</code></i>	Display current clock values, such as FPS.

```
pyglet.clock.Clock
```

Clock Class

class `Clock` (*fps_limit=None, time_function=<built-in function time>*)

Class for calculating and limiting framerate, and for calling scheduled functions.

Attributes:

<i><code>MIN_SLEEP</code></i>	The minimum amount of time in seconds this clock will attempt to sleep for when framerate limiting.
<i><code>SLEEP_UNDERSHOOT</code></i>	The amount of time in seconds this clock subtracts from sleep values to compensate for lazy operating system.

Attributes

`Clock.MIN_SLEEP = 0.005`

The minimum amount of time in seconds this clock will attempt to sleep for when framerate limiting. Higher values will increase the accuracy of the limiting but also increase CPU usage while busy-waiting. Lower values mean the process sleeps more often, but is prone to over-sleep and run at a potentially lower or uneven framerate than desired.

`Clock.SLEEP_UNDERSHOOT = 0.004`

The amount of time in seconds this clock subtracts from sleep values to compensate for lazy operating systems.

pyglet.clock.ClockDisplay

ClockDisplay Class

class `ClockDisplay` (*font=None*, *interval=0.25*, *format='%(fps).2f'*, *color=(0.5, 0.5, 0.5, 0.5)*,
clock=None)

Display current clock values, such as FPS.

This is a convenience class for displaying diagnostics such as the framerate. See the module documentation for example usage.

Variables `label` – The label which is displayed.

Warning: Deprecated. This class presents values that are often misleading, as they reflect the rate of clock ticks, not displayed framerate. Use `pyglet.window.FPSDisplay` instead.

Functions

<code>get_default()</code>	Return the <i>Clock</i> instance that is used by all module-level clock functions.
<code>get_fps()</code>	Return the current measured FPS of the default clock.
<code>get_fps_limit()</code>	Get the framerate limit for the default clock.
<code>get_sleep_time(sleep_idle)</code>	Get the time until the next item is scheduled on the default clock.
<code>schedule(func, *args, **kwargs)</code>	Schedule 'func' to be called every frame on the default clock.
<code>schedule_interval(func, interval, *args, ...)</code>	Schedule 'func' to be called every 'interval' seconds on the default clock.
<code>schedule_interval_soft(func, interval, ...)</code>	Schedule 'func' to be called every 'interval' seconds on the default clock, beginning at the next frame.
<code>schedule_once(func, delay, *args, **kwargs)</code>	Schedule 'func' to be called once after 'delay' seconds (can be a float) on the default clock.
<code>set_default(default)</code>	Set the default clock to use for all module-level functions.
<code>set_fps_limit(fps_limit)</code>	Set the framerate limit for the default clock.
<code>test_clock()</code>	
<code>tick([poll])</code>	Signify that one frame has passed on the default clock.
<code>unschedule(func)</code>	Remove 'func' from the default clock's schedule.

***get_default* Function** Defined in `pyglet.clock`

`get_default` ()

Return the *Clock* instance that is used by all module-level clock functions.

Return type *Clock*

Returns The default clock.

***get_fps* Function** Defined in *pyglet.clock*

***get_fps* ()**

Return the current measured FPS of the default clock.

Return type float

***get_fps_limit* Function** Defined in *pyglet.clock*

***get_fps_limit* ()**

Get the framerate limit for the default clock.

Returns The framerate limit previously set by *set_fps_limit*, or None if no limit was set.

***get_sleep_time* Function** Defined in *pyglet.clock*

***get_sleep_time* (*sleep_idle*)**

Get the time until the next item is scheduled on the default clock.

See *Clock.get_sleep_time* for details.

Parameters ***sleep_idle*** (*bool*) – If True, the application intends to sleep through its idle time; otherwise it will continue ticking at the maximum frame rate allowed.

Return type float

Returns Time until the next scheduled event in seconds, or None if there is no event scheduled.

Note: Since pyglet 1.1

***schedule* Function** Defined in *pyglet.clock*

***schedule* (*func*, **args*, ***kwargs*)**

Schedule ‘func’ to be called every frame on the default clock.

The arguments passed to func are dt, followed by any **args* and ***kwargs* given here.

Parameters ***func*** (*function*) – The function to call each frame.

***schedule_interval* Function** Defined in *pyglet.clock*

***schedule_interval* (*func*, *interval*, **args*, ***kwargs*)**

Schedule ‘func’ to be called every ‘interval’ seconds on the default clock.

The arguments passed to ‘func’ are ‘dt’ (time since last function call), followed by any **args* and ***kwargs* given here.

Parameters

- ***func*** (*function*) – The function to call when the timer lapses.
- ***interval*** (*float*) – The number of seconds to wait between each call.

***schedule_interval_soft* Function** Defined in *pyglet.clock*

schedule_interval_soft (*func*, *interval*, **args*, ***kwargs*)

Schedule 'func' to be called every 'interval' seconds on the default clock, beginning at a time that does not coincide with other scheduled events.

The arguments passed to 'func' are 'dt' (time since last function call), followed by any *args and **kwargs given here.

See *Clock.schedule_interval_soft*

Note: Since pyglet 1.1

Parameters

- **func** (*function*) – The function to call when the timer lapses.
- **interval** (*float*) – The number of seconds to wait between each call.

***schedule_once* Function** Defined in *pyglet.clock*

schedule_once (*func*, *delay*, **args*, ***kwargs*)

Schedule 'func' to be called once after 'delay' seconds (can be a float) on the default clock. The arguments passed to 'func' are 'dt' (time since last function call), followed by any *args and **kwargs given here.

If no default clock is set, the func is queued and will be scheduled on the default clock as soon as it is created.

Parameters

- **func** (*function*) – The function to call when the timer lapses.
- **delay** (*float*) – The number of seconds to wait before the timer lapses.

***set_default* Function** Defined in *pyglet.clock*

set_default (*default*)

Set the default clock to use for all module-level functions.

By default an instance of *Clock* is used.

Parameters **default** (*Clock*) – The default clock to use.

***set_fps_limit* Function** Defined in *pyglet.clock*

set_fps_limit (*fps_limit*)

Set the framerate limit for the default clock.

Parameters **fps_limit** (*float*) – Maximum frames per second allowed, or None to disable limiting.

Warning: Deprecated. Use *pyglet.app.run* and *schedule_interval* instead.

***test_clock* Function** Defined in *pyglet.clock*

test_clock ()

tick Function Defined in `pyglet.clock`

tick (*poll=False*)

Signify that one frame has passed on the default clock.

This will call any scheduled functions that have elapsed.

Parameters **poll** (*bool*) – If True, the function will call any scheduled functions but will not sleep or busy-wait for any reason. Recommended for advanced applications managing their own sleep timers only. Since pyglet 1.1.

Return type float

Returns The number of seconds since the last “tick”, or 0 if this was the first frame.

unschedule Function Defined in `pyglet.clock`

unschedule (*func*)

Remove ‘func’ from the default clock’s schedule. No error is raised if the func was never scheduled.

Parameters **func** (*function*) – The function to remove from the schedule.

Variables

compat_platform = ‘linux’

`str(object=’’) -> str str(bytes_or_buffer[, encoding[, errors]]) -> str`

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of `object.__str__()` (if defined) or `repr(object)`. encoding defaults to `sys.getdefaultencoding()`. errors defaults to ‘strict’.

division = `_Feature((2, 2, 0, ‘alpha’, 2), (3, 0, 0, ‘alpha’, 0), 8192)`

print_function = `_Feature((2, 6, 0, ‘alpha’, 2), (3, 0, 0, ‘alpha’, 0), 65536)`

Notes

Defined

- `pyglet`
- `time`

`pyglet.debug`

Functions

`debug_print([enabled_or_option])` Get a debug printer that is enabled based on a boolean input or a pyglet option.

debug_print Function Defined in `pyglet.debug`

debug_print (*enabled_or_option='debug'*)

Get a debug printer that is enabled based on a boolean input or a pyglet option. The debug print function returned

should be used in an assert. This way it can be optimized out when running python with the -O flag.

Usage example:

```
from pyglet.debug import debug_print
_debug_media = debug_print('debug_media')

def some_func():
    assert _debug_media('My debug statement')
```

Parameters *enabled_or_options* (*bool* or *str*) – If a bool is passed, debug printing is enabled if it is True. If str is passed debug printing is enabled if the pyglet option with that name is True.

Returns Function for debug printing.

Variables

`print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)`

Notes

Defined

- `pyglet`

pyglet.event

Event dispatch framework.

All objects that produce events in pyglet implement *EventDispatcher*, providing a consistent interface for registering and manipulating event handlers. A commonly used event dispatcher is *pyglet.window.Window*.

Event types

For each event dispatcher there is a set of events that it dispatches; these correspond with the type of event handlers you can attach. Event types are identified by their name, for example, “on_resize”. If you are creating a new class which implements *EventDispatcher*, you must call *EventDispatcher.register_event_type* for each event type.

Attaching event handlers

An event handler is simply a function or method. You can attach an event handler by setting the appropriate function on the instance:

```
def on_resize(width, height):
    # ...
dispatcher.on_resize = on_resize
```

There is also a convenience decorator that reduces typing:


```
@dispatcher.event
def on_resize(width, height):
    # ...
```

You may prefer to subclass and override the event handlers instead:

```
class MyDispatcher(DispatcherClass):
    def on_resize(self, width, height):
        # ...
```

Event handler stack

When attaching an event handler to a dispatcher using the above methods, it replaces any existing handler (causing the original handler to no longer be called). Each dispatcher maintains a stack of event handlers, allowing you to insert an event handler “above” the existing one rather than replacing it.

There are two main use cases for “pushing” event handlers:

- Temporarily intercepting the events coming from the dispatcher by pushing a custom set of handlers onto the dispatcher, then later “popping” them all off at once.
- Creating “chains” of event handlers, where the event propagates from the top-most (most recently added) handler to the bottom, until a handler takes care of it.

Use *EventDispatcher.push_handlers* to create a new level in the stack and attach handlers to it. You can push several handlers at once:

```
dispatcher.push_handlers(on_resize, on_key_press)
```

If your function handlers have different names to the events they handle, use keyword arguments:

```
dispatcher.push_handlers(on_resize=my_resize,
                        on_key_press=my_key_press)
```

After an event handler has processed an event, it is passed on to the next-lowest event handler, unless the handler returns *EVENT_HANDLED*, which prevents further propagation.

To remove all handlers on the top stack level, use *EventDispatcher.pop_handlers*.

Note that any handlers pushed onto the stack have precedence over the handlers set directly on the instance (for example, using the methods described in the previous section), regardless of when they were set. For example, handler `foo` is called before handler `bar` in the following example:

```
dispatcher.push_handlers(on_resize=foo)
dispatcher.on_resize = bar
```

Dispatching events

pyglet uses a single-threaded model for all application code. Event handlers are only ever invoked as a result of calling *EventDispatcher.dispatch_events*.

It is up to the specific event dispatcher to queue relevant events until they can be dispatched, at which point the handlers are called in the order the events were originally generated.

This implies that your application runs with a main loop that continuously updates the application state and checks for new events:

```
while True:
    dispatcher.dispatch_events()
    # ... additional per-frame processing
```

Not all event dispatchers require the call to `dispatch_events`; check with the particular class documentation.

Classes

EventDispatcher Generic event dispatcher interface.

pyglet.event.EventDispatcher

EventDispatcher Class

class **EventDispatcher**

Generic event dispatcher interface.

See the module docstring for usage.

Methods:

register_event_type(name) Register an event type with the dispatcher.

Methods

classmethod **EventDispatcher.register_event_type**(name)

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters **name** (*str*) – Name of the event to register.

Exceptions

EventException An exception raised when an event handler could not be attached.

pyglet.event.EventException

EventException

Exception defined in `pyglet.event`

exception EventException

An exception raised when an event handler could not be attached.

Variables**EVENT_HANDLED = True**

`bool(x) -> bool`

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

EVENT_UNHANDLED = None**Notes****Defined**

- inspect

`pyglet.font`

Load fonts and render text.

This is a fairly-low level interface to text rendering. Obtain a font using `load()`:

```
from pyglet import font
arial = font.load('Arial', 14, bold=True, italic=False)
```

Manually loading fonts is only required in the following situations:

- When manually rendering fonts;
- When using the deprecated font rendering in `pyglet.font.text`.

You are encouraged to use `pyglet.text` for actual text rendering. Classes in this module will handle font loading for you, so manual loading is not required.

pyglet will automatically load any system-installed fonts. You can add additional fonts (for example, from your program resources) using `add_file()` or `add_directory()`. These fonts are then available in the same way as system-installed fonts:

```
from pyglet import font
font.add_file('action_man.ttf')
action_man = font.load('Action Man', 16)
```

See the `pyglet.font.base` module for documentation on the base classes used by this package.

Modules

<code>base</code>	Abstract classes used by <code>pyglet.font</code> implementations.
<code>text</code>	Deprecated text rendering
<code>ttf</code>	Implementation of the TrueType file format.

`pyglet.font.base` Abstract classes used by `pyglet.font` implementations.

These classes should not be constructed directly. Instead, use the functions in `pyglet.font` to obtain platform-specific instances. You can use these classes as a documented interface to the concrete classes.

<code>Font</code>	Abstract font class able to produce glyphs.
<code>Glyph</code>	A single glyph located within a larger texture.
<code>GlyphRenderer</code>	Abstract class for creating glyph images.
<code>GlyphTextureAtlas</code>	A texture within which glyphs can be drawn.

Classes

`pyglet.font.base.Font`

Font Class

class `Font`

Abstract font class able to produce glyphs.

To construct a font, use `pyglet.font.load`, which will instantiate the platform-specific font class.

Internally, this class is used by the platform classes to manage the set of textures into which glyphs are written.

Variables

- **`ascent`** – Maximum ascent above the baseline, in pixels.
- **`descent`** – Maximum descent below the baseline, in pixels. Usually negative.

Methods:

<code>add_font_data(data)</code>	Add font data to the font loader.
<code>have_font(name)</code>	Determine if a font with the given name is installed.

Attributes:

`ascent`

Continued on next page

Table 2.25 – continued from previous page

<code>descent</code>
<code>texture_height</code>
<code>texture_internalformat</code>
<code>texture_width</code>

Methods

classmethod `Font.add_font_data(data)`

Add font data to the font loader.

This is a class method and affects all fonts loaded. Data must be some byte string of data, for example, the contents of a TrueType font file. Subclasses can override this method to add the font data into the font registry.

There is no way to instantiate a font given the data directly, you must use `pyglet.font.load` specifying the font name.

classmethod `Font.have_font(name)`

Determine if a font with the given name is installed.

Parameters `name` (*str*) – Name of a font to search for

Return type `bool`

Attributes

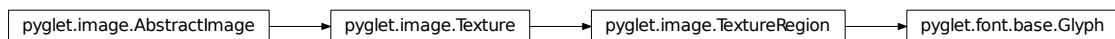
`Font.ascent = 0`

`Font.descent = 0`

`Font.texture_height = 256`

`Font.texture_internalformat = 6406`

`Font.texture_width = 256`

**Glyph Class**

class `Glyph(x, y, z, width, height, owner)`

A single glyph located within a larger texture.

Glyphs are drawn most efficiently using the higher level APIs, for example `GlyphString`.

Variables

- **`advance`** – The horizontal advance of this glyph, in pixels.
- **`vertices`** – The vertices of this glyph, with (0,0) originating at the left-side bearing at the baseline.

Methods:

Attributes:

<i>advance</i>	
anchor_x	
anchor_y	
image_data	An ImageData view of this texture.
images	
level	
mipmapped_texture	A Texture view of this image.
tex_coords	
tex_coords_order	
texture	Get a <i>Texture</i> view of this image.
<i>vertices</i>	
x	
y	
z	

Attributes

`Glyph.advance = 0`

`Glyph.vertices = (0, 0, 0, 0)`

Inherited members**Methods**

`Glyph.create` (*width*, *height*, *internalformat*=6408, *rectangle*=False, *force_rectangle*=False, *min_filter*=9729, *mag_filter*=9729)

Create an empty Texture.

If *rectangle* is False or the appropriate driver extensions are not available, a larger texture than requested will be created, and a *TextureRegion* corresponding to the requested size will be returned.

Parameters

- **width** (*int*) – Width of the texture.
- **height** (*int*) – Height of the texture.
- **internalformat** (*int*) – GL constant giving the internal format of the texture; for example, GL_RGBA.
- **rectangle** (*bool*) – True if a rectangular texture is permitted. See *AbstractImage.get_texture*.
- **force_rectangle** (*bool*) – True if a rectangular texture is required. See *AbstractImage.get_texture*. **Since:** pyglet 1.1.4.
- **min_filter** (*int*) – The minification filter used for this texture, commonly GL_LINEAR or GL_NEAREST
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly GL_LINEAR or GL_NEAREST

Return type *Texture*

Note: Since pyglet 1.1

`Glyph.create_for_size` (*target*, *min_width*, *min_height*, *internalformat=None*,
min_filter=9729, *mag_filter=9729*)

Create a Texture with dimensions at least *min_width*, *min_height*. On return, the texture will be bound.

Parameters

- **target** (*int*) – GL constant giving texture target to use, typically `GL_TEXTURE_2D`.
- **min_width** (*int*) – Minimum width of texture (may be increased to create a power of 2).
- **min_height** (*int*) – Minimum height of texture (may be increased to create a power of 2).
- **internalformat** (*int*) – GL constant giving internal format of texture; for example, `GL_RGBA`. If unspecified, the texture will not be initialised (only the texture name will be created on the instance). If specified, the image will be initialised to this format with zero'd data.
- **min_filter** (*int*) – The minification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`

Return type *Texture*

Attributes

`Glyph.anchor_x = 0`

`Glyph.anchor_y = 0`

`Glyph.image_data`

An *ImageData* view of this texture.

Changes to the returned instance will not be reflected in this texture. If the texture is a 3D texture, the first image will be returned. See also *get_image_data*. Read-only.

Warning: Deprecated. Use *get_image_data*.

Type *ImageData*

`Glyph.images = 1`

`Glyph.level = 0`

`Glyph.mipmapped_texture`

A *Texture* view of this image.

The returned *Texture* will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use *get_mipmapped_texture*.

Type *Texture*

`Glyph.tex_coords = (0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0)`

`Glyph.tex_coords_order = (0, 1, 2, 3)`

`Glyph.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*

`Glyph.x = 0`

`Glyph.y = 0`

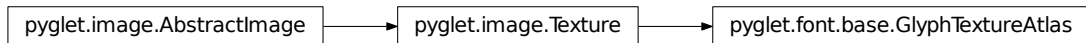
`Glyph.z = 0`

`pyglet.font.base.GlyphRenderer`

GlyphRenderer Class

class `GlyphRenderer` (*font*)

Abstract class for creating glyph images.



GlyphTextureAtlas Class

class `GlyphTextureAtlas` (*width, height, target, id*)

A texture within which glyphs can be drawn.

Methods:

Attributes:

`anchor_x`

`anchor_y`

`image_data` An *ImageData* view of this texture.

`images`

`level`

Continued on next page

Table 2.29 – continued from previous page

<code>line_height</code>	
<code>mipmapped_texture</code>	A Texture view of this image.
<code>tex_coords</code>	
<code>tex_coords_order</code>	
<code>texture</code>	Get a <i>Texture</i> view of this image.
<code>x</code>	
<code>y</code>	
<code>z</code>	

Attributes

`GlyphTextureAtlas.line_height = 0`

`GlyphTextureAtlas.x = 0`

`GlyphTextureAtlas.y = 0`

Inherited members**Methods**

`GlyphTextureAtlas.create`(*width*, *height*, *internalformat*=6408, *rectangle*=False, *force_rectangle*=False, *min_filter*=9729, *mag_filter*=9729)

Create an empty Texture.

If *rectangle* is False or the appropriate driver extensions are not available, a larger texture than requested will be created, and a *TextureRegion* corresponding to the requested size will be returned.

Parameters

- **width** (*int*) – Width of the texture.
- **height** (*int*) – Height of the texture.
- **internalformat** (*int*) – GL constant giving the internal format of the texture; for example, GL_RGBA.
- **rectangle** (*bool*) – True if a rectangular texture is permitted. See *AbstractImage.get_texture*.
- **force_rectangle** (*bool*) – True if a rectangular texture is required. See *AbstractImage.get_texture*. **Since:** pyglet 1.1.4.
- **min_filter** (*int*) – The minification filter used for this texture, commonly GL_LINEAR or GL_NEAREST
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly GL_LINEAR or GL_NEAREST

Return type *Texture*

Note: Since pyglet 1.1

`GlyphTextureAtlas.create_for_size(target, min_width, min_height, internalformat=None, min_filter=9729, mag_filter=9729)`

Create a Texture with dimensions at least `min_width`, `min_height`. On return, the texture will be bound.

Parameters

- **target** (*int*) – GL constant giving texture target to use, typically `GL_TEXTURE_2D`.
- **min_width** (*int*) – Minimum width of texture (may be increased to create a power of 2).
- **min_height** (*int*) – Minimum height of texture (may be increased to create a power of 2).
- **internalformat** (*int*) – GL constant giving internal format of texture; for example, `GL_RGBA`. If unspecified, the texture will not be initialised (only the texture name will be created on the instance). If specified, the image will be initialised to this format with zero'd data.
- **min_filter** (*int*) – The minification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`

Return type *Texture*

Attributes

`GlyphTextureAtlas.anchor_x = 0`

`GlyphTextureAtlas.anchor_y = 0`

`GlyphTextureAtlas.image_data`

An `ImageData` view of this texture.

Changes to the returned instance will not be reflected in this texture. If the texture is a 3D texture, the first image will be returned. See also `get_image_data`. Read-only.

Warning: Deprecated. Use `get_image_data`.

Type *ImageData*

`GlyphTextureAtlas.images = 1`

`GlyphTextureAtlas.level = 0`

`GlyphTextureAtlas.mipmapped_texture`

A `Texture` view of this image.

The returned `Texture` will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use `get_mipmapped_texture`.

Type *Texture*

`GlyphTextureAtlas.tex_coords = (0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0)`

`GlyphTextureAtlas.tex_coords_order = (0, 1, 2, 3)`

`GlyphTextureAtlas.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*

`GlyphTextureAtlas.z = 0`

FontException Generic exception related to errors from the font module.

Exceptions

`pyglet.font.base.FontException`

FontException

Exception defined in `pyglet.font.base`

exception FontException

Generic exception related to errors from the font module. Typically these relate to invalid font data.

`get_grapheme_clusters(text)` Implements Table 2 of UAX #29: Grapheme Cluster Boundaries.

Functions

`get_grapheme_clusters` Function Defined in `pyglet.font.base`

`get_grapheme_clusters` (*text*)

Implements Table 2 of UAX #29: Grapheme Cluster Boundaries.

Does not currently implement Hangul syllable rules.

Parameters **text** (*unicode*) – String to cluster.

Note: Since pyglet 1.1.2

Return type List of *unicode*

Returns List of Unicode grapheme clusters

Variables

`absolute_import` = `_Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)`

```
compat_platform = 'linux'
```

```
str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
```

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

```
print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)
```

Defined

Notes

- gl
- gltext_arb
- glu
- image
- lib
- lib_glx
- unicodedata

pyglet.font.text Deprecated text rendering

This is a fairly-low level interface to text rendering. Obtain a font using *load*:

```
from pyglet import font
arial = font.load('Arial', 14, bold=True, italic=False)
```

pyglet will load any system-installed fonts. You can add additional fonts (for example, from your program resources) using *add_file* or *add_directory*.

Obtain a list of *Glyph* objects for a string of text using the *Font* object:

```
text = 'Hello, world!'
glyphs = arial.get_glyphs(text)
```

The most efficient way to render these glyphs is with a *GlyphString*:

```
glyph_string = GlyphString(text, glyphs)
glyph_string.draw()
```

There are also a variety of methods in both *Font* and *GlyphString* to facilitate word-wrapping.

A convenient way to render a string of text is with a *Text*:

```
text = Text(font, text)
text.draw()
```

See the *pyglet.font.base* module for documentation on the base classes used by this package.

<i>GlyphString</i>	An immutable string of glyphs that can be rendered quickly.
<i>Text</i>	Simple displayable text.

Classes

pyglet.font.text.GlyphString

GlyphString Class

class **GlyphString** (*text, glyphs, x=0, y=0*)

An immutable string of glyphs that can be rendered quickly.

This class is ideal for quickly rendering single or multi-line strings of text that use the same font. To wrap text using a glyph string, call *get_break_index* to find the optimal breakpoint for each line, then repeatedly call *draw* for each breakpoint.

Warning: Deprecated. Use *pyglet.text.layout* classes.

pyglet.font.text.Text

Text Class

class **Text** (*font, text='', x=0, y=0, z=0, color=(1, 1, 1, 1), width=None, halign='left', valign='baseline'*)

Simple displayable text.

This is a convenience class for rendering strings of text. It takes care of caching the vertices so the text can be rendered every frame with little performance penalty.

Text can be word-wrapped by specifying a *width* to wrap into. If the width is not specified, it gives the width of the text as laid out.

Variables

- *x* – X coordinate of the text
- *y* – Y coordinate of the text

Warning: Deprecated. Use *pyglet.text.Label*.

Attributes:

<i>BASELINE</i>	Align the baseline of the first line of text with the given Y coordinate.
<i>BOTTOM</i>	Align the bottom of the descender of the final line of text with the given Y coordinate.
<i>CENTER</i>	Align the horizontal center of the text to the given X coordinate.
<i>LEFT</i>	Align the left edge of the text to the given X coordinate.
<i>RIGHT</i>	Align the right edge of the text to the given X coordinate.
<i>TOP</i>	Align the top of the ascender of the first line of text with the given Y coordinate.
<i>color</i>	
<i>font</i>	

Continued on next page

Table 2.33 – continued from previous page

<i>halign</i>	Horizontal alignment of the text.
<i>height</i>	Height of the text.
<i>leading</i>	Vertical space between adjacent lines, in pixels.
<i>line_height</i>	Vertical distance between adjacent baselines, in pixels.
<i>text</i>	Text to render.
<i>valign</i>	Vertical alignment of the text.
<i>width</i>	Width of the text.
<i>x</i>	
<i>y</i>	
<i>z</i>	

Attributes

`Text.BASELINE = 'baseline'`

Align the baseline of the first line of text with the given Y coordinate.

`Text.BOTTOM = 'bottom'`

Align the bottom of the descender of the final line of text with the given Y coordinate.

`Text.CENTER = 'center'`

Align the horizontal center of the text to the given X coordinate.

`Text.LEFT = 'left'`

Align the left edge of the text to the given X coordinate.

`Text.RIGHT = 'right'`

Align the right edge of the text to the given X coordinate.

`Text.TOP = 'top'`

Align the top of the ascender of the first line of text with the given Y coordinate.

`Text.color`

`Text.font`

`Text.halign`

Horizontal alignment of the text.

The text is positioned relative to *x* and *width* according to this property, which must be one of the alignment constants *LEFT*, *CENTER* or *RIGHT*.

Type str

`Text.height`

Height of the text.

This property is the ascent minus the descent of the font, unless there is more than one line of word-wrapped text, in which case the height takes into account the line leading. Read-only.

Type float

`Text.leading`

Vertical space between adjacent lines, in pixels.

Type int

`Text.line_height`

Vertical distance between adjacent baselines, in pixels.

Type int

`Text.text`

Text to render.

The glyph vertices are only recalculated as needed, so multiple changes to the text can be performed with no performance penalty.

Type str

`Text.valign`

Vertical alignment of the text.

The text is positioned relative to y according to this property, which must be one of the alignment constants *BOTTOM*, *BASELINE*, *CENTER* or *TOP*.

Type str

`Text.width`

Width of the text.

When set, this enables word-wrapping to the specified width. Otherwise, the width of the text as it will be rendered can be determined.

Type float

`Text.x`

`Text.y`

`Text.z`

Variables

absolute_import = `_Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)`

compat_platform = `'linux'`

`str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str`

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of `object.__str__()` (if defined) or `repr(object)`. encoding defaults to `sys.getdefaultencoding()`. errors defaults to `'strict'`.

division = `_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)`

print_function = `_Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)`

Defined

Notes

- `gl`
- `glctx_arb`
- `glu`
- `lib`
- `lib_glx`
- `pyglet`
- `warnings`

pyglet.font.ttf Implementation of the TrueType file format.

Typical applications will not need to use this module directly; look at *pyglyph.font* instead.

References:

- <http://developer.apple.com/fonts/TTRefMan/RM06>
- <http://www.microsoft.com/typography/otspec>

TrueTypeInfo Information about a single TrueType face.

Classes

pyglet.font.ttf.TruetypeInfo

TrueTypeInfo Class

class TruetypeInfo (*filename*)

Information about a single TrueType face.

The class memory-maps the font file to read the tables, so it is vital that you call the *close* method to avoid large memory leaks. Once closed, you cannot call any of the *get_** methods.

Not all tables have been implemented yet (or likely ever will). Currently only the name and metric tables are read; in particular there is no glyph or hinting information.

Variables

division = `_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)`

Defined

Notes

- codecs
- mmap
- os
- struct

Functions

<i>add_directory</i> (dir)	Add a directory of fonts to pyglet's search path.
<i>add_file</i> (font)	Add a font to pyglet's search path.
<i>have_font</i> (name)	Check if specified system font name is available.
<i>load</i> ([name, size, bold, italic, dpi])	Load a font for rendering.

add_directory Function Defined in `pyglet.font`

add_directory (*dir*)

Add a directory of fonts to pyglet’s search path.

This function simply calls `pyglet.font.add_file()` for each file with a `.ttf` extension in the given directory. Subdirectories are not searched.

Parameters **dir** (*str*) – Directory that contains font files.

add_file Function Defined in `pyglet.font`

add_file (*font*)

Add a font to pyglet’s search path.

In order to load a font that is not installed on the system, you must call this method to tell pyglet that it exists. You can supply either a filename or any file-like object.

The font format is platform-dependent, but is typically a TrueType font file containing a single font face. Note that to use a font added with this method, you should pass the face name (not the file name) to `pyglet.font.load()` or any other place where you normally specify a font.

Parameters **font** (*str* or *file*) – Filename or file-like object to load fonts from.

have_font Function Defined in `pyglet.font`

have_font (*name*)

Check if specified system font name is available.

load Function Defined in `pyglet.font`

load (*name=None, size=None, bold=False, italic=False, dpi=None*)

Load a font for rendering.

Parameters

- **name** (*str*, or *list of str*) – Font family, for example, “Times New Roman”. If a list of names is provided, the first one matching a known font is used. If no font can be matched to the name(s), a default font is used. In pyglet 1.1, the name may be omitted.
- **size** (*float*) – Size of the font, in points. The returned font may be an exact match or the closest available. In pyglet 1.1, the size may be omitted, and defaults to 12pt.
- **bold** (*bool*) – If True, a bold variant is returned, if one exists for the given family and size.
- **italic** (*bool*) – If True, an italic variant is returned, if one exists for the given family and size.
- **dpi** (*float*) – The assumed resolution of the display device, for the purposes of determining the pixel size of the font. Defaults to 96.

Return type *Font*

Variables

absolute_import = `_Feature((2, 5, 0, ‘alpha’, 1), (3, 0, 0, ‘alpha’, 0), 16384)`

division = `_Feature((2, 2, 0, ‘alpha’, 2), (3, 0, 0, ‘alpha’, 0), 8192)`

gl = <module 'pyglet.gl' from '/home/docs/checkouts/readthedocs.org/user_builds/pyglet/checkouts/latest/pyglet/gl/__init__.py'
OpenGL and GLU interface.

This package imports all OpenGL, GLU and registered OpenGL extension functions. Functions have identical signatures to their C counterparts. For example:

```
from pyglet.gl import *

# [...omitted: set up a GL context and framebuffer]
glBegin(GL_QUADS)
glVertex3f(0, 0, 0)
glVertex3f(0.1, 0.2, 0.3)
glVertex3f(0.1, 0.2, 0.3)
glVertex3f(0.1, 0.2, 0.3)
glEnd()
```

OpenGL is documented in full at the [OpenGL Reference Pages](#).

The [OpenGL Programming Guide](#) is a popular reference manual organised by topic. The free online version documents only OpenGL 1.1. [Later editions](#) cover more recent versions of the API and can be purchased from a book store.

The following subpackages are imported into this “mega” package already (and so are available by importing `pyglet.gl`):

`pyglet.gl.gl` OpenGL

`pyglet.gl.glu` GLU

`pyglet.gl.gl.glext_arb` ARB registered OpenGL extension functions

These subpackages are also available, but are not imported into this namespace by default:

`pyglet.gl.glext_nv` nVidia OpenGL extension functions

`pyglet.gl.agl` AGL (Mac OS X OpenGL context functions)

`pyglet.gl.glx` GLX (Linux OpenGL context functions)

`pyglet.gl.glxext_arb` ARB registered GLX extension functions

`pyglet.gl.glxext_nv` nvidia GLX extension functions

`pyglet.gl.wgl` WGL (Windows OpenGL context functions)

`pyglet.gl.wglext_arb` ARB registered WGL extension functions

`pyglet.gl.wglext_nv` nvidia WGL extension functions

The information modules are provided for convenience, and are documented below.

Notes

Defined

- `os`
- `pyglet`
- `sys`
- `weakref`

pyglet.gl

OpenGL and GLU interface.

This package imports all OpenGL, GLU and registered OpenGL extension functions. Functions have identical signatures to their C counterparts. For example:

```
from pyglet.gl import *

# [...omitted: set up a GL context and framebuffer]
glBegin(GL_QUADS)
glVertex3f(0, 0, 0)
glVertex3f(0.1, 0.2, 0.3)
glVertex3f(0.1, 0.2, 0.3)
glVertex3f(0.1, 0.2, 0.3)
glEnd()
```

OpenGL is documented in full at the [OpenGL Reference Pages](#).

The [OpenGL Programming Guide](#) is a popular reference manual organised by topic. The free online version documents only OpenGL 1.1. [Later editions](#) cover more recent versions of the API and can be purchased from a book store.

The following subpackages are imported into this “mega” package already (and so are available by importing `pyglet.gl`):

pyglet.gl.gl OpenGL

pyglet.gl.glu GLU

pyglet.gl.gl.glext_arb ARB registered OpenGL extension functions

These subpackages are also available, but are not imported into this namespace by default:

pyglet.gl.glext_nv nVidia OpenGL extension functions

pyglet.gl.agl AGL (Mac OS X OpenGL context functions)

pyglet.gl.glx GLX (Linux OpenGL context functions)

pyglet.gl.glxext_arb ARB registered GLX extension functions

pyglet.gl.glxext_nv nvidia GLX extension functions

pyglet.gl.wgl WGL (Windows OpenGL context functions)

pyglet.gl.wglext_arb ARB registered WGL extension functions

pyglet.gl.wglext_nv nvidia WGL extension functions

The information modules are provided for convenience, and are documented below.

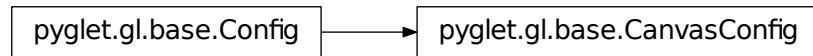
Modules

<i>base</i>	
<i>gl</i>	Wrapper for /usr/include/GL/gl.h
<i>gl_info</i>	Information about version and extensions of current GL implementation.
<i>glu</i>	Wrapper for /usr/include/GL/glu.h
<i>glu_info</i>	Information about version and extensions of current GLU implementation.
<i>lib</i>	

pyglet.gl.base

<i>CanvasConfig</i>	OpenGL configuration for a particular canvas.
<i>Config</i>	Graphics configuration.
<i>Context</i>	OpenGL context for drawing.
<i>ObjectSpace</i>	

Classes



CanvasConfig Class

class CanvasConfig (*canvas*, *base_config*)
 OpenGL configuration for a particular canvas.
 Use *Config.match* to obtain an instance of this class.

Note: Since pyglet 1.2

Variables *canvas* – The canvas this config is valid on.

Attributes:

<i>debug</i>
<i>forward_compatible</i>
<i>major_version</i>
<i>minor_version</i>

Inherited members

Attributes

`CanvasConfig.debug = None`
`CanvasConfig.forward_compatible = None`
`CanvasConfig.major_version = None`
`CanvasConfig.minor_version = None`

pyglet.gl.base.Config

Config Class**class Config** (***kwargs*)

Graphics configuration.

A Config stores the preferences for OpenGL attributes such as the number of auxilliary buffers, size of the colour and depth buffers, double buffering, stencilling, multi- and super-sampling, and so on.

Different platforms support a different set of attributes, so these are set with a string key and a value which is integer or boolean.

Variables

- **double_buffer** – Specify the presence of a back-buffer for every color buffer.
- **stereo** – Specify the presence of separate left and right buffer sets.
- **buffer_size** – Total bits per sample per color buffer.
- **aux_buffers** – The number of auxilliary color buffers.
- **sample_buffers** – The number of multisample buffers.
- **samples** – The number of samples per pixel, or 0 if there are no multisample buffers.
- **red_size** – Bits per sample per buffer devoted to the red component.
- **green_size** – Bits per sample per buffer devoted to the green component.
- **blue_size** – Bits per sample per buffer devoted to the blue component.
- **alpha_size** – Bits per sample per buffer devoted to the alpha component.
- **depth_size** – Bits per sample in the depth buffer.
- **stencil_size** – Bits per sample in the stencil buffer.
- **accum_red_size** – Bits per pixel devoted to the red component in the accumulation buffer.
- **accum_green_size** – Bits per pixel devoted to the green component in the accumulation buffer.
- **accum_blue_size** – Bits per pixel devoted to the blue component in the accumulation buffer.
- **accum_alpha_size** – Bits per pixel devoted to the alpha component in the accumulation buffer.

Attributes:

<i>debug</i>
<i>forward_compatible</i>
<i>major_version</i>
<i>minor_version</i>

Attributes`Config.debug = None``Config.forward_compatible = None``Config.major_version = None``Config.minor_version = None`

pyglet.gl.base.Context

Context Class**class Context** (*config, context_share=None*)

OpenGL context for drawing.

Use *CanvasConfig.create_context* to create a context.**Variables** `object_space` – An object which is shared between all contexts that share GL objects.**Attributes:**

<code>CONTEXT_SHARE_EXISTING</code>	Context share behaviour indicating that objects are shared with the most recently created context (the default).
<code>CONTEXT_SHARE_NONE</code>	Context share behaviour indicating that objects should not be shared with existing contexts.

Attributes`Context.CONTEXT_SHARE_EXISTING = 1`

Context share behaviour indicating that objects are shared with the most recently created context (the default).

`Context.CONTEXT_SHARE_NONE = None`

Context share behaviour indicating that objects should not be shared with existing contexts.

pyglet.gl.base.ObjectSpace

ObjectSpace Class**class ObjectSpace****Variables**`compat_platform = 'linux'``str(object='') -> str(str(bytes_or_buffer[, encoding[, errors]]) -> str`

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of

object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

gl = <pyglet._ModuleProxy object>

pyglet.gl.gl Wrapper for /usr/include/GL/gl.h

Generated by tools/gengl.py. Do not modify this file.

Variables

DEFAULT_MODE = 0

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

Notes	Defined
	• util

pyglet.gl.gl_info Information about version and extensions of current GL implementation.

Usage:

```
from pyglet.gl import gl_info

if gl_info.have_extension('GL_NV_register_combiners'):
    # ...
```

If you are using more than one context, you can set up a separate GLInfo object for each context. Call *set_active_context* after switching to the context:

```
from pyglet.gl.gl_info import GLInfo

info = GLInfo()
info.set_active_context()

if info.have_version(2, 1):
    # ...
```

GLInfo Information interface for a single GL context.

Classes

pyglet.gl.gl_info.GLInfo

***GLInfo* Class**

class GLInfo

Information interface for a single GL context.

A default instance is created automatically when the first OpenGL context is created. You can use the module functions as a convenience for this default instance's methods.

If you are using more than one context, you must call *set_active_context* when the context is active for this *GLInfo* instance.

Attributes:

<i>extensions</i>
<i>have_context</i>
<i>renderer</i>
<i>vendor</i>
<i>version</i>

Attributes

```
GLInfo.extensions = set()
GLInfo.have_context = False
GLInfo.renderer = ''
GLInfo.vendor = ''
GLInfo.version = '0.0.0'
```

<i>have_context()</i>	Determine if a default OpenGL context has been set yet.
-----------------------	---

Functions

***have_context* Function** Defined in *pyglet.gl.gl_info*

have_context()

Determine if a default OpenGL context has been set yet.

Return type bool

Variables

get_extensions = <bound method GLInfo.get_extensions of <pyglet.gl.gl_info.GLInfo object at 0x7fa5508b5a90>>

Get a list of available OpenGL extensions.

Returns a list of the available extensions.

Return type list of str

get_renderer = <bound method GLInfo.get_renderer of <pyglet.gl.gl_info.GLInfo object at 0x7fa5508b5a90>>
Determine the renderer string of the OpenGL context.

Return type str

get_vendor = <bound method GLInfo.get_vendor of <pyglet.gl.gl_info.GLInfo object at 0x7fa5508b5a90>>
Determine the vendor string of the OpenGL context.

Return type str

get_version = <bound method GLInfo.get_version of <pyglet.gl.gl_info.GLInfo object at 0x7fa5508b5a90>>
Get the current OpenGL version.

Returns the OpenGL version

Return type str

have_extension = <bound method GLInfo.have_extension of <pyglet.gl.gl_info.GLInfo object at 0x7fa5508b5a90>>
Determine if an OpenGL extension is available.

Parameters **extension** (*str*) – The name of the extension to test for, including its GL_ prefix.

Returns True if the extension is provided by the driver.

Return type bool

have_version = <bound method GLInfo.have_version of <pyglet.gl.gl_info.GLInfo object at 0x7fa5508b5a90>>
Determine if a version of OpenGL is supported.

Parameters

- **major** (*int*) – The major revision number (typically 1 or 2).
- **minor** (*int*) – The minor revision number.
- **release** (*int*) – The release number.

Return type bool

Returns True if the requested or a later version is supported.

remove_active_context = <bound method GLInfo.remove_active_context of <pyglet.gl.gl_info.GLInfo object at 0x7fa5508b5a90>>

set_active_context = <bound method GLInfo.set_active_context of <pyglet.gl.gl_info.GLInfo object at 0x7fa5508b5a90>>

Store information for the currently active context.

This method is called automatically for the default context.

Defined

Notes

- util
- warnings

pyglet.gl.glu Wrapper for /usr/include/GL/glu.h

Generated by tools/gengl.py. Do not modify this file.

pyglet.gl.glu_info Information about version and extensions of current GLU implementation.

Usage:

```
from pyglet.gl import glu_info

if glu_info.have_extension('GLU_EXT_nurbs_tessellator'):
    # ...
```

If multiple contexts are in use you can use a separate `GLUInfo` object for each context. Call `set_active_context` after switching to the desired context for each `GLUInfo`:

```
from pyglet.gl.glu_info import GLUInfo

info = GLUInfo()
info.set_active_context()
if info.have_version(1, 3):
    # ...
```

Note that `GLUInfo` only returns meaningful information if a context has been created.

GLUInfo Information interface for the GLU library.

Classes

pyglet.gl.glu_info.GLUInfo

GLUInfo Class

class **GLUInfo**

Information interface for the GLU library.

A default instance is created automatically when the first OpenGL context is created. You can use the module functions as a convenience for this default instance's methods.

If you are using more than one context, you must call `set_active_context` when the context is active for this *GLUInfo* instance.

Attributes:

extensions

have_context

version

Attributes

```
GLUInfo.extensions = []
GLUInfo.have_context = False
```

`GLUInfo.version = '0.0.0'`

Variables

get_extensions = <bound method GLUInfo.get_extensions of <pyglet.gl.glu_info.GLUInfo object at 0x7fa5508c3048>>

Get a list of available GLU extensions.

Returns a list of the available extensions.

Return type list of str

get_version = <bound method GLUInfo.get_version of <pyglet.gl.glu_info.GLUInfo object at 0x7fa5508c3048>>

Get the current GLU version.

Returns the GLU version

Return type str

have_extension = <bound method GLUInfo.have_extension of <pyglet.gl.glu_info.GLUInfo object at 0x7fa5508c3048>>

Determine if a GLU extension is available.

Parameters **extension** (*str*) – The name of the extension to test for, including its GLU_ prefix.

Returns True if the extension is provided by the implementation.

Return type bool

have_version = <bound method GLUInfo.have_version of <pyglet.gl.glu_info.GLUInfo object at 0x7fa5508c3048>>

Determine if a version of GLU is supported.

Parameters

- **major** (*int*) – The major revision number (typically 1).
- **minor** (*int*) – The minor revision number.
- **release** (*int*) – The release number.

Return type bool

Returns True if the requested or a later version is supported.

set_active_context = <bound method GLUInfo.set_active_context of <pyglet.gl.glu_info.GLUInfo object at 0x7fa5508c3048>>

Store information for the currently active context.

This method is called automatically for the default context.

Defined

Notes

- util
- warnings

`pyglet.gl.lib`

`c_void`

Classes



c_void Class

class **c_void**

Attributes:

<i>dummy</i>	Type: CField
--------------	--------------

Attributes

`c_void.dummy`

Structure/Union member

GLEException

MissingFunctionException(name[, requires, ...])

Exceptions

pyglet.gl.lib.GLEException

GLEException

Exception defined in *pyglet.gl.lib*

exception **GLEException**

pyglet.gl.lib.MissingFunctionException

MissingFunctionException

Exception defined in *pyglet.gl.lib*

exception **MissingFunctionException** (name, requires=None, suggestions=None)

```

    decorate_function(func, name)
    errcheck(result, func, arguments)
    errcheck_glbeg(result, func, arguments)
    errcheck_glend(result, func, arguments)
    missing_function(name[, requires, suggestions])

```

Functions

***decorate_function* Function** Defined in *pyglet.gl.lib*

decorate_function (*func*, *name*)

***errcheck* Function** Defined in *pyglet.gl.lib*

errcheck (*result*, *func*, *arguments*)

***errcheck_glbeg* Function** Defined in *pyglet.gl.lib*

errcheck_glbeg (*result*, *func*, *arguments*)

***errcheck_glend* Function** Defined in *pyglet.gl.lib*

errcheck_glend (*result*, *func*, *arguments*)

***missing_function* Function** Defined in *pyglet.gl.lib*

missing_function (*name*, *requires*=None, *suggestions*=None)

Variables

link_AGL = None

link_WGL = None

print_function = *_Feature*((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)

Notes	Defined
	<ul style="list-style-type: none"> pyglet

Classes

Exceptions

```

    ConfigException
    ContextException

```

pyglet.gl.ConfigException

ConfigException

Exception defined in *pyglet.gl*

exception `ConfigException`

pyglet.gl.ContextException

ContextException

Exception defined in *pyglet.gl*

exception `ContextException`

Functions

get_current_context() Return the active OpenGL context.

***get_current_context* Function** Defined in *pyglet.gl*

`get_current_context()`

Return the active OpenGL context.

You can change the current context by calling *Context.set_current*.

Warning: Deprecated. Use *current_context*

Return type *Context*

Returns the context to which OpenGL commands are directed, or None if there is no selected context.

Variables

`absolute_import` = `_Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)`

`compat_platform` = `'linux'`

`str(object='') -> str` `str(bytes_or_buffer[, encoding[, errors]]) -> str`

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of

object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

```
print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)
```

Notes

Defined

- glxext_arb
- lib_glx

pyglet.graphics

Low-level graphics rendering.

This module provides an efficient low-level abstraction over OpenGL. It gives very good performance for rendering OpenGL primitives; far better than the typical immediate-mode usage and, on modern graphics cards, better than using display lists in many cases. The module is used internally by other areas of pyglet.

See the Programming Guide for details on how to use this graphics API.

Batches and groups

Without even needing to understand the details on how to draw primitives with the graphics API, developers can make use of *Batch* and *Group* objects to improve performance of sprite and text rendering.

The *Sprite*, *Label* and *TextLayout* classes all accept a *batch* and *group* parameter in their constructors. A batch manages a set of objects that will be drawn all at once, and a group describes the manner in which an object is drawn.

The following example creates a batch, adds two sprites to the batch, and then draws the entire batch:

```
batch = pyglet.graphics.Batch()
car = pyglet.sprite.Sprite(car_image, batch=batch)
boat = pyglet.sprite.Sprite(boat_image, batch=batch)

def on_draw():
    batch.draw()
```

Drawing a complete batch is much faster than drawing the items in the batch individually, especially when those items belong to a common group.

Groups describe the OpenGL state required for an item. This is for the most part managed by the sprite and text classes, however you can also use groups to ensure items are drawn in a particular order. For example, the following example adds a background sprite which is guaranteed to be drawn before the car and the boat:

```
batch = pyglet.graphics.Batch()
background = pyglet.graphics.OrderedGroup(0)
foreground = pyglet.graphics.OrderedGroup(1)

background = pyglet.sprite.Sprite(background_image,
                                   batch=batch, group=background)
car = pyglet.sprite.Sprite(car_image, batch=batch, group=foreground)
boat = pyglet.sprite.Sprite(boat_image, batch=batch, group=foreground)
```

```
def on_draw()
    batch.draw()
```

It's preferable to manage sprites and text objects within as few batches as possible. If the drawing of sprites or text objects need to be interleaved with other drawing that does not use the graphics API, multiple batches will be required.

Data item parameters

Many of the functions and methods in this module accept any number of data parameters as their final parameters. In the documentation these are notated as **data* in the formal parameter list.

A data parameter describes a vertex attribute format and an optional sequence to initialise that attribute. Examples of common attribute formats are:

"v3f" Vertex position, specified as three floats.

"c4B" Vertex color, specified as four unsigned bytes.

"t2f" Texture coordinate, specified as two floats.

See *pyglet.graphics.vertexattribute* for the complete syntax of the vertex format string.

When no initial data is to be given, the data item is just the format string. For example, the following creates a 2 element vertex list with position and color attributes:

```
vertex_list = pyglet.graphics.vertex_list(2, 'v2f', 'c4B')
```

When initial data is required, wrap the format string and the initial data in a tuple, for example:

```
vertex_list = pyglet.graphics.vertex_list(2,
                                           ('v2f', (0.0, 1.0, 1.0, 0.0)),
                                           ('c4B', (255, 255, 255, 255) * 2))
```

Drawing modes

Methods in this module that accept a mode parameter will accept any value in the OpenGL drawing mode enumeration: `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_QUAD_STRIP`, `GL_QUADS`, and `GL_POLYGON`.

```
pyglet.graphics.draw(1, GL_POINTS, ('v2i', (10, 20)))
```

However, because of the way the graphics API renders multiple primitives with shared state, `GL_POLYGON`, `GL_LINE_LOOP` and `GL_TRIANGLE_FAN` cannot be used — the results are undefined.

When using `GL_LINE_STRIP`, `GL_TRIANGLE_STRIP` or `GL_QUAD_STRIP` care must be taken to insert degenerate vertices at the beginning and end of each vertex list. For example, given the vertex list:

```
A, B, C, D
```

the correct vertex list to provide the vertex list is:

```
A, A, B, C, D, D
```

Alternatively, the `NV_primitive_restart` extension can be used if it is present. This also permits use of `GL_POLYGON`, `GL_LINE_LOOP` and `GL_TRIANGLE_FAN`. Unfortunately the extension is not provided by older video drivers, and requires indexed vertex lists.

Note: Since pyglet 1.1

Modules

<i>allocation</i>	Memory allocation algorithm for vertex arrays and buffers.
<i>vertexattribute</i>	Access byte arrays as arrays of vertex attributes.
<i>vertexbuffer</i>	Byte abstractions of Vertex Buffer Objects and vertex arrays.
<i>vertexdomain</i>	Manage related vertex attributes within a single vertex domain.

pyglet.graphics.allocation Memory allocation algorithm for vertex arrays and buffers.

The region allocator is used to allocate vertex indices within a vertex domain's multiple buffers. ("Buffer" refers to any abstract buffer presented by *pyglet.graphics.vertexbuffer*.

The allocator will at times request more space from the buffers. The current policy is to double the buffer size when there is not enough room to fulfil an allocation. The buffer is never resized smaller.

The allocator maintains references to free space only; it is the caller's responsibility to maintain the allocated regions.

<i>Allocator</i>	Buffer space allocation implementation.
------------------	---

Classes

pyglet.graphics.allocation.Allocator

Allocator Class

class *Allocator* (*capacity*)
Buffer space allocation implementation.

<i>AllocatorMemoryException</i> (requested_capacity)	The buffer is not large enough to fulfil an allocation.
--	---

Exceptions

pyglet.graphics.allocation.AllocatorMemoryException

AllocatorMemoryException

Exception defined in `pyglet.graphics.allocation`

exception AllocatorMemoryException (*requested_capacity*)

The buffer is not large enough to fulfil an allocation.

Raised by *Allocator* methods when the operation failed due to lack of buffer space. The buffer should be increased to at least `requested_capacity` and then the operation retried (guaranteed to pass second time).

Variables

division = `_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)`

print_function = `_Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)`

pyglet.graphics.vertexattribute Access byte arrays as arrays of vertex attributes.

Use *create_attribute* to create an attribute accessor given a simple format string. Alternatively, the classes may be constructed directly.

Attribute format strings An attribute format string specifies the format of a vertex attribute. Format strings are accepted by the *create_attribute* function as well as most methods in the *pyglet.graphics* module.

Format strings have the following (BNF) syntax:

attribute ::= (name | index 'g' 'n'? | texture 't') count type

`name` describes the vertex attribute, and is one of the following constants for the predefined attributes:

c Vertex color

e Edge flag

f Fog coordinate

n Normal vector

s Secondary color

t Texture coordinate

v Vertex coordinate

You can alternatively create a generic indexed vertex attribute by specifying its index in decimal followed by the constant `g`. For example, `0g` specifies the generic vertex attribute with index 0. If the optional constant `n` is present after the `g`, the attribute is normalised to the range `[0, 1]` or `[-1, 1]` within the range of the data type.

Texture coordinates for multiple texture units can be specified with the texture number before the constant `t`. For example, `1t` gives the texture coordinate attribute for texture unit 1.

`count` gives the number of data components in the attribute. For example, a 3D vertex position has a count of 3. Some attributes constrain the possible counts that can be used; for example, a normal vector must have a count of 3.

`type` gives the data type of each component of the attribute. The following types can be used:

b GLbyte
B GLubyte
s GLshort
S GLushort
i GLint
I GLuint
f GLfloat
d GLdouble

Some attributes constrain the possible data types; for example, normal vectors must use one of the signed data types. The use of some data types, while not illegal, may have severe performance concerns. For example, the use of `GLdouble` is discouraged, and colours should be specified with `GLubyte`.

Whitespace is prohibited within the format string.

Some examples follow:

v3f 3-float vertex position

c4b 4-byte colour

1eb Edge flag

0g3f 3-float generic vertex attribute 0

1gn1i Integer generic vertex attribute 1, normalized to [-1, 1]

2gn4B 4-byte generic vertex attribute 2, normalized to [0, 1] (because the type is unsigned)

3t2f 2-float texture coordinate for texture unit 3.

<i>AbstractAttribute</i>	Abstract accessor for an attribute in a mapped buffer.
<i>ColorAttribute</i>	Color vertex attribute.
<i>EdgeFlagAttribute</i>	Edge flag attribute.
<i>FogCoordAttribute</i>	Fog coordinate attribute.
<i>GenericAttribute</i>	Generic vertex attribute, used by shader programs.
<i>MultiTexCoordAttribute</i>	Texture coordinate attribute.
<i>NormalAttribute</i>	Normal vector attribute.
<i>SecondaryColorAttribute</i>	Secondary color attribute.
<i>TexCoordAttribute</i>	Texture coordinate attribute.
<i>VertexAttribute</i>	Vertex coordinate attribute.

Classes

pyglet.graphics.vertexattribute.AbstractAttribute

AbstractAttribute Class

class AbstractAttribute (*count, gl_type*)
 Abstract accessor for an attribute in a mapped buffer.



ColorAttribute Class

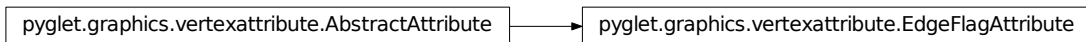
class ColorAttribute (*count, gl_type*)
 Color vertex attribute.

Attributes:

plural

Attributes

ColorAttribute.**plural** = 'colors'



EdgeFlagAttribute Class

class EdgeFlagAttribute (*gl_type*)
 Edge flag attribute.

Attributes:

plural

Attributes

EdgeFlagAttribute.**plural** = 'edge_flags'



FogCoordAttribute Class

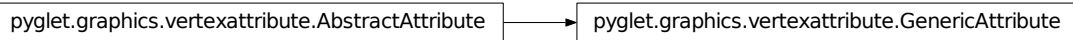
class FogCoordAttribute (*count, gl_type*)
Fog coordinate attribute.

Attributes:

plural

Attributes

FogCoordAttribute.**plural** = 'fog_coords'



GenericAttribute Class

class GenericAttribute (*index, normalized, count, gl_type*)
Generic vertex attribute, used by shader programs.



MultiTexCoordAttribute Class

class MultiTexCoordAttribute (*texture, count, gl_type*)
Texture coordinate attribute.



NormalAttribute Class

class NormalAttribute (*gl_type*)
Normal vector attribute.

Attributes:

plural

Attributes

`NormalAttribute.plural = 'normals'`



SecondaryColorAttribute Class

class **SecondaryColorAttribute** (*gl_type*)
 Secondary color attribute.

Attributes:

plural

Attributes

`SecondaryColorAttribute.plural = 'secondary_colors'`



TexCoordAttribute Class

class **TexCoordAttribute** (*count, gl_type*)
 Texture coordinate attribute.

Attributes:

plural

Attributes

`TexCoordAttribute.plural = 'tex_coords'`



VertexAttribute Class

class **VertexAttribute** (*count, gl_type*)
 Vertex coordinate attribute.

Attributes:*plural***Attributes**VertexAttribute.**plural** = 'vertices'

<i>create_attribute</i> (format)	Create a vertex attribute description from a format string.
<i>interleave_attributes</i> (attributes)	Interleave attribute offsets.
<i>serialize_attributes</i> (count, attributes)	Serialize attribute offsets.

Functions**create_attribute Function** Defined in *pyglet.graphics.vertexattribute***create_attribute** (format)

Create a vertex attribute description from a format string.

The initial stride and offset of the attribute will be 0.

Parameters **format** (*str*) – Attribute format string. See the module summary for details.**Return type** *AbstractAttribute***interleave_attributes Function** Defined in *pyglet.graphics.vertexattribute***interleave_attributes** (attributes)

Interleave attribute offsets.

Adjusts the offsets and strides of the given attributes so that they are interleaved. Alignment constraints are respected.

Parameters **attributes** (*sequence of AbstractAttribute*) – Attributes to interleave in-place.**serialize_attributes Function** Defined in *pyglet.graphics.vertexattribute***serialize_attributes** (count, attributes)

Serialize attribute offsets.

Adjust the offsets of the given attributes so that they are packed serially against each other for *count* vertices.**Parameters**

- **count** (*int*) – Number of vertices.
- **attributes** (*sequence of AbstractAttribute*) – Attributes to serialize in-place.

Variables

```
absolute_import = _Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)
```

```
compat_platform = 'linux'
```

```
str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str
```

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

```
print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)
```

Defined

- Notes**
- gl
 - glext_arb
 - glu
 - lib
 - lib_glx
 - re
 - vertexbuffer

pyglet.graphics.vertexbuffer Byte abstractions of Vertex Buffer Objects and vertex arrays.

Use *create_buffer* or *create_mappable_buffer* to create a Vertex Buffer Object, or a vertex array if VBOs are not supported by the current context.

Buffers can optionally be created “mappable” (incorporating the *AbstractMappable* mix-in). In this case the buffer provides a *get_region* method which provides the most efficient path for updating partial data within the buffer.

<i>AbstractBuffer</i>	Abstract buffer of byte data.
<i>AbstractBufferRegion</i>	A mapped region of a buffer.
<i>AbstractMappable</i>	
<i>IndirectArrayRegion</i>	A mapped region in which data elements are not necessarily contiguous.
<i>MappableVertexBufferObject</i>	A VBO with system-memory backed store.
<i>VertexArray</i>	A ctypes implementation of a vertex array.
<i>VertexArrayRegion</i>	A mapped region of a vertex array.
<i>VertexBufferObject</i>	Lightweight representation of an OpenGL VBO.
<i>VertexBufferObjectRegion</i>	A mapped region of a VBO.

Classes

pyglet.graphics.vertexbuffer.AbstractBuffer

AbstractBuffer Class

class AbstractBuffer

Abstract buffer of byte data.

Variables

- *size* – Size of buffer, in bytes
- *ptr* – Memory offset of the buffer, as used by the `glVertexPointer` family of functions
- *target* – OpenGL buffer target, for example `GL_ARRAY_BUFFER`
- *usage* – OpenGL buffer usage, for example `GL_DYNAMIC_DRAW`

Attributes:

ptr

size

Attributes

`AbstractBuffer.ptr = 0`

`AbstractBuffer.size = 0`

pyglet.graphics.vertexbuffer.AbstractBufferRegion

AbstractBufferRegion Class**class AbstractBufferRegion**

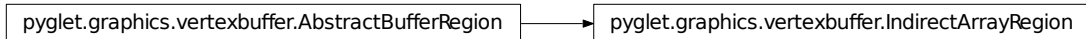
A mapped region of a buffer.

Buffer regions are obtained using *AbstractMappable.get_region*.

Variables *array* – Array of data, of the type and count requested by *get_region*.

pyglet.graphics.vertexbuffer.AbstractMappable

AbstractMappable Class**class AbstractMappable**

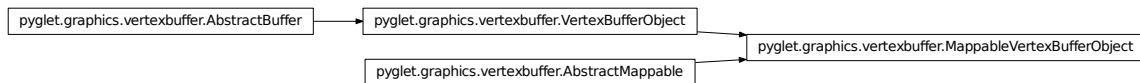


IndirectArrayRegion Class

class IndirectArrayRegion (*region, size, component_count, component_stride*)

A mapped region in which data elements are not necessarily contiguous.

This region class is used to wrap buffer regions in which the data must be accessed with some stride. For example, in an interleaved buffer this region can be used to access a single interleaved component as if the data was contiguous.



MappableVertexBufferObject Class

class MappableVertexBufferObject (*size, target, usage*)

A VBO with system-memory backed store.

Updates to the data via *set_data*, *set_data_region* and *map* will be held in local memory until *bind* is called. The advantage is that fewer OpenGL calls are needed, increasing performance.

There may also be less performance penalty for resizing this buffer.

Updates to data via *map* are committed immediately.

Attributes:

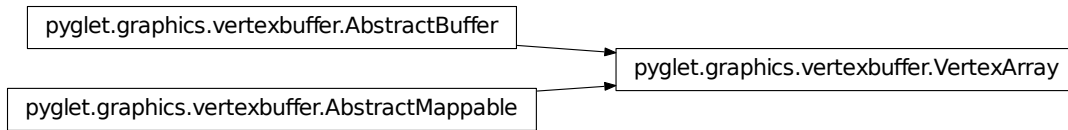
<code>ptr</code>
<code>size</code>

Inherited members

Attributes

`MappableVertexBufferObject.ptr = 0`

`MappableVertexBufferObject.size = 0`

**VertexArray Class**

class **VertexArray** (*size*)

A ctypes implementation of a vertex array.

Many of the methods on this class are effectively no-op's, such as *bind*, *unbind*, *map*, *unmap* and *delete*; they exist in order to present a consistent interface with *VertexBufferObject*.

This buffer type is also mappable, and so *get_region* can be used.

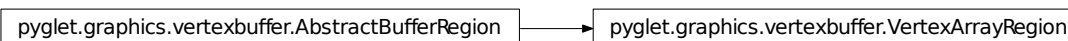
Attributes:

ptr
size

Inherited members**Attributes**

`VertexArray.ptr = 0`

`VertexArray.size = 0`

**VertexArrayRegion Class**

class **VertexArrayRegion** (*array*)

A mapped region of a vertex array.

The *invalidate* method is a no-op but is provided in order to present a consistent interface with *VertexBufferObjectRegion*.

pyglet.graphics.vertexbuffer.AbstractBuffer

pyglet.graphics.vertexbuffer.VertexBufferObject

VertexBufferObject Class

class **VertexBufferObject** (*size, target, usage*)

Lightweight representation of an OpenGL VBO.

The data in the buffer is not replicated in any system memory (unless it is done so by the video driver). While this can improve memory usage and possibly performance, updates to the buffer are relatively slow.

This class does not implement *AbstractMappable*, and so has no `get_region` method. See *MappableVertexBufferObject* for a VBO class that does implement `get_region`.

Attributes:

`ptr`

`size`

Inherited members

Attributes

`VertexBufferObject.ptr = 0`

`VertexBufferObject.size = 0`

pyglet.graphics.vertexbuffer.AbstractBufferRegion

pyglet.graphics.vertexbuffer.VertexBufferObjectRegion

VertexBufferObjectRegion Class

class **VertexBufferObjectRegion** (*buffer, start, end, array*)

A mapped region of a VBO.

`create_buffer`(*size[, target, usage, vbo]*) Create a buffer of vertex data.

`create_mappable_buffer`(*size[, target, ...]*) Create a mappable buffer of vertex data.

Functions

`create_buffer` Function Defined in `pyglet.graphics.vertexbuffer`

`create_buffer` (*size, target=34962, usage=35048, vbo=True*)

Create a buffer of vertex data.

Parameters

- **size** (*int*) – Size of the buffer, in bytes
- **target** (*int*) – OpenGL target buffer
- **usage** (*int*) – OpenGL usage constant
- **vbo** (*bool*) – True if a *VertexBufferObject* should be created if the driver supports it; otherwise only a *VertexArray* is created.

Return type *AbstractBuffer*

create_mappable_buffer Function Defined in *pyglet.graphics.vertexbuffer*

create_mappable_buffer (*size*, *target*=34962, *usage*=35048, *vbo*=True)

Create a mappable buffer of vertex data.

Parameters

- **size** (*int*) – Size of the buffer, in bytes
- **target** (*int*) – OpenGL target buffer
- **usage** (*int*) – OpenGL usage constant
- **vbo** (*bool*) – True if a *VertexBufferObject* should be created if the driver supports it; otherwise only a *VertexArray* is created.

Return type *AbstractBuffer* with *AbstractMappable*

Variables

absolute_import = *_Feature*((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)

compat_platform = 'linux'

str(object='') -> *str* *str*(bytes_or_buffer[, encoding[, errors]]) -> *str*

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of *object.__str__*() (if defined) or *repr*(object). encoding defaults to *sys.getdefaultencoding*(). errors defaults to 'strict'.

print_function = *_Feature*((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)

Defined

- *gl*
- *glctx_arb*
- *glu*
- *lib*
- *lib_glx*
- *pyglet*
- *sys*

Notes

pyglet.graphics.vertexdomain Manage related vertex attributes within a single vertex domain.

A vertex “domain” consists of a set of attribute descriptions that together describe the layout of one or more vertex buffers which are used together to specify the vertices in a primitive. Additionally, the domain manages the buffers used to store the data and will resize them as necessary to accommodate new vertices.

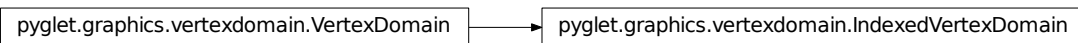
Domains can optionally be indexed, in which case they also manage a buffer containing vertex indices. This buffer is grown separately and has no size relation to the attribute buffers.

Applications can create vertices (and optionally, indices) within a domain with the *VertexDomain.create* method. This returns a *VertexList* representing the list of vertices created. The vertex attribute data within the group can be modified, and the changes will be made to the underlying buffers automatically.

The entire domain can be efficiently drawn in one step with the *VertexDomain.draw* method, assuming all the vertices comprise primitives of the same OpenGL primitive mode.

<i>IndexedVertexDomain</i>	Management of a set of indexed vertex lists.
<i>IndexedVertexList</i>	A list of vertices within an <i>IndexedVertexDomain</i> that are indexed.
<i>VertexDomain</i>	Management of a set of vertex lists.
<i>VertexList</i>	A list of vertices within a <i>VertexDomain</i> .

Classes

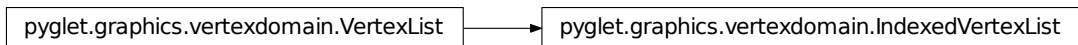


IndexedVertexDomain Class

class IndexedVertexDomain (*attribute_usages*, *index_gl_type*=5125)

Management of a set of indexed vertex lists.

Construction of an indexed vertex domain is usually done with the *create_indexed_domain* function.



IndexedVertexList Class

class IndexedVertexList (*domain*, *start*, *count*, *index_start*, *index_count*)

A list of vertices within an *IndexedVertexDomain* that are indexed. Use *IndexedVertexDomain.create* to construct this list.

Attributes:

<i>colors</i>	Array of color data.
<i>edge_flags</i>	Array of edge flag data.
<i>fog_coords</i>	Array of fog coordinate data.
Continued on next page	

Table 2.72 – continued from previous page

<i>indices</i>	Array of index data.
<i>multi_tex_coords</i>	Multi-array texture coordinate data.
<i>normals</i>	Array of normal vector data.
<i>secondary_colors</i>	Array of secondary color data.
<i>tex_coords</i>	Array of texture coordinate data.
<i>vertices</i>	Array of vertex coordinate data.

Attributes`IndexedVertexList.indices`

Array of index data.

Inherited members**Attributes**`IndexedVertexList.colors`

Array of color data.

`IndexedVertexList.edge_flags`

Array of edge flag data.

`IndexedVertexList.fog_coords`

Array of fog coordinate data.

`IndexedVertexList.multi_tex_coords`

Multi-array texture coordinate data.

`IndexedVertexList.normals`

Array of normal vector data.

`IndexedVertexList.secondary_colors`

Array of secondary color data.

`IndexedVertexList.tex_coords`

Array of texture coordinate data.

`IndexedVertexList.vertices`

Array of vertex coordinate data.

`pyglet.graphics.vertexdomain.VertexDomain`

VertexDomain Class**class** `VertexDomain` (*attribute_usages*)

Management of a set of vertex lists.

Construction of a vertex domain is usually done with the *create_domain* function.

pyglet.graphics.vertexdomain.VertexList

VertexList Class

class VertexList (*domain, start, count*)

A list of vertices within a *VertexDomain*. Use *VertexDomain.create* to construct this list.

Attributes:

<i>colors</i>	Array of color data.
<i>edge_flags</i>	Array of edge flag data.
<i>fog_coords</i>	Array of fog coordinate data.
<i>multi_tex_coords</i>	Multi-array texture coordinate data.
<i>normals</i>	Array of normal vector data.
<i>secondary_colors</i>	Array of secondary color data.
<i>tex_coords</i>	Array of texture coordinate data.
<i>vertices</i>	Array of vertex coordinate data.

Attributes

`VertexList.colors`

Array of color data.

`VertexList.edge_flags`

Array of edge flag data.

`VertexList.fog_coords`

Array of fog coordinate data.

`VertexList.multi_tex_coords`

Multi-array texture coordinate data.

`VertexList.normals`

Array of normal vector data.

`VertexList.secondary_colors`

Array of secondary color data.

`VertexList.tex_coords`

Array of texture coordinate data.

`VertexList.vertices`

Array of vertex coordinate data.

<i>create_attribute_usage</i> (format)	Create an attribute and usage pair from a format string.
<i>create_domain</i> (*attribute_usage_formats)	Create a vertex domain covering the given attribute usage formats.
<i>create_indexed_domain</i> (*attribute_usage_formats)	Create an indexed vertex domain covering the given attribute usage form

Functions

create_attribute_usage Function Defined in `pyglet.graphics.vertexdomain`

create_attribute_usage (*format*)

Create an attribute and usage pair from a format string. The format string is as documented in `pyglet.graphics.vertexattribute`, with the addition of an optional usage component:

```
usage ::= attribute ( '/' ('static' | 'dynamic' | 'stream' | 'none') )?
```

If the usage is not given it defaults to 'dynamic'. The usage corresponds to the OpenGL VBO usage hint, and for `static` also indicates a preference for interleaved arrays. If `none` is specified a buffer object is not created, and vertex data is stored in system memory.

Some examples:

v3f/stream 3D vertex position using floats, for stream usage

c4b/static 4-byte color attribute, for static usage

Returns attribute, usage

create_domain Function Defined in `pyglet.graphics.vertexdomain`

create_domain (**attribute_usage_formats*)

Create a vertex domain covering the given attribute usage formats. See documentation for `create_attribute_usage` and `pyglet.graphics.vertexattribute.create_attribute` for the grammar of these format strings.

Return type `VertexDomain`

create_indexed_domain Function Defined in `pyglet.graphics.vertexdomain`

create_indexed_domain (**attribute_usage_formats*)

Create an indexed vertex domain covering the given attribute usage formats. See documentation for `create_attribute_usage` and `pyglet.graphics.vertexattribute.create_attribute` for the grammar of these format strings.

Return type `VertexDomain`

Variables

absolute_import = `_Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)`

compat_platform = `'linux'`

`str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str`

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of `object.__str__()` (if defined) or `repr(object)`. encoding defaults to `sys.getdefaultencoding()`. errors defaults to 'strict'.

print_function = `_Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)`

Defined

- Notes**
- allocation
 - gl
 - glext_arb
 - glu
 - lib
 - lib_glx
 - re
 - vertexattribute
 - vertexbuffer

Classes

<i>Batch</i>	Manage a collection of vertex lists for batched rendering.
<i>Group</i>	Group of common OpenGL state.
<i>NullGroup</i>	The default group class used when <code>None</code> is given to a batch.
<i>OrderedGroup</i>	A group with partial order.
<i>TextureGroup</i>	A group that enables and binds a texture.

pyglet.graphics.Batch

Batch Class**class Batch**

Manage a collection of vertex lists for batched rendering.

Vertex lists are added to a *Batch* using the *add* and *add_indexed* methods. An optional group can be specified along with the vertex list, which gives the OpenGL state required for its rendering. Vertex lists with shared mode and group are allocated into adjacent areas of memory and sent to the graphics card in a single operation.

Call *VertexList.delete* to remove a vertex list from the batch.

pyglet.graphics.Group

Group Class**class Group** (*parent=None*)

Group of common OpenGL state.

Before a vertex list is rendered, its group's OpenGL state is set; as are that state's ancestors' states. This can be

defined arbitrarily on subclasses; the default state change has no effect, and groups vertex lists only in the order in which they are drawn.



NullGroup Class

class NullGroup (*parent=None*)

The default group class used when `None` is given to a batch.

This implementation has no effect.

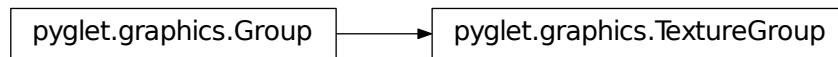


OrderedGroup Class

class OrderedGroup (*order, parent=None*)

A group with partial order.

Ordered groups with a common parent are rendered in ascending order of their `order` field. This is a useful way to render multiple layers of a scene within a single batch.



TextureGroup Class

class TextureGroup (*texture, parent=None*)

A group that enables and binds a texture.

Texture groups are equal if their textures' targets and names are equal.

Functions

<code>draw(size, mode, *data)</code>	Draw a primitive immediately.
--------------------------------------	-------------------------------

Continued on next page

Table 2.76 – continued from previous page

<code>draw_indexed(size, mode, indices, *data)</code>	Draw a primitive with indexed vertices immediately.
<code>vertex_list(count, *data)</code>	Create a <i>VertexList</i> not associated with a batch, group or mode.
<code>vertex_list_indexed(count, indices, *data)</code>	Create an <i>IndexedVertexList</i> not associated with a batch, group or mode.

draw Function Defined in `pyglet.graphics`

draw (*size, mode, *data*)

Draw a primitive immediately.

Parameters

- **size** (*int*) – Number of vertices given
- **mode** (*gl primitive type*) – OpenGL drawing mode, e.g. `GL_TRIANGLES`, avoiding quotes.
- **data** (*data items*) – Attribute formats and data. See the module summary for details.

draw_indexed Function Defined in `pyglet.graphics`

draw_indexed (*size, mode, indices, *data*)

Draw a primitive with indexed vertices immediately.

Parameters

- **size** (*int*) – Number of vertices given
- **mode** (*int*) – OpenGL drawing mode, e.g. `GL_TRIANGLES`
- **indices** (*sequence of int*) – Sequence of integers giving indices into the vertex list.
- **data** (*data items*) – Attribute formats and data. See the module summary for details.

vertex_list Function Defined in `pyglet.graphics`

vertex_list (*count, *data*)

Create a *VertexList* not associated with a batch, group or mode.

Parameters

- **count** (*int*) – The number of vertices in the list.
- **data** (*data items*) – Attribute formats and initial data for the vertex list. See the module summary for details.

Return type *VertexList*

vertex_list_indexed Function Defined in `pyglet.graphics`

vertex_list_indexed (*count, indices, *data*)

Create an *IndexedVertexList* not associated with a batch, group or mode.

Parameters

- **count** (*int*) – The number of vertices in the list.
- **indices** (*sequence*) – Sequence of integers giving indices into the vertex list.
- **data** (*data items*) – Attribute formats and initial data for the vertex list. See the module summary for details.

Return type *IndexedVertexList*

Variables

absolute_import = *_Feature*((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)

compat_platform = 'linux'

`str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str`

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of `object.__str__()` (if defined) or `repr(object)`. encoding defaults to `sys.getdefaultencoding()`. errors defaults to 'strict'.

null_group = *<pyglet.graphics.NullGroup object>*

The default group.

Type *Group*

print_function = *_Feature*((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)

Notes

Defined

- `gl`
- `glext_arb`
- `glu`
- `lib`
- `lib_glx`
- `pyglet`

pyglet.image

Image load, capture and high-level texture functions.

Only basic functionality is described here; for full reference see the accompanying documentation.

To load an image:

```
from pyglet import image
pic = image.load('picture.png')
```

The supported image file types include PNG, BMP, GIF, JPG, and many more, somewhat depending on the operating system. To load an image from a file-like object instead of a filename:

```
pic = image.load('hint.jpg', file=fileobj)
```

The hint helps the module locate an appropriate decoder to use based on the file extension. It is optional.

Once loaded, images can be used directly by most other modules of pyglet. All images have a width and height you can access:

```
width, height = pic.width, pic.height
```

You can extract a region of an image (this keeps the original image intact; the memory is shared efficiently):

```
subimage = pic.get_region(x, y, width, height)
```

Remember that y-coordinates are always increasing upwards.

Drawing images

To draw an image at some point on the screen:

```
pic.blit(x, y, z)
```

This assumes an appropriate view transform and projection have been applied.

Some images have an intrinsic “anchor point”: this is the point which will be aligned to the x and y coordinates when the image is drawn. By default the anchor point is the lower-left corner of the image. You can use the anchor point to center an image at a given point, for example:

```
pic.anchor_x = pic.width // 2
pic.anchor_y = pic.height // 2
pic.blit(x, y, z)
```

Texture access

If you are using OpenGL directly, you can access the image as a texture:

```
texture = pic.get_texture()
```

(This is the most efficient way to obtain a texture; some images are immediately loaded as textures, whereas others go through an intermediate form). To use a texture with pyglet.gl:

```
from pyglet.gl import *
glEnable(texture.target)          # typically target is GL_TEXTURE_2D
glBindTexture(texture.target, texture.id)
# ... draw with the texture
```

Pixel access

To access raw pixel data of an image:

```
rawimage = pic.get_image_data()
```

(If the image has just been loaded this will be a very quick operation; however if the image is a texture a relatively expensive readback operation will occur). The pixels can be accessed as a string:

```
format = 'RGBA'
pitch = rawimage.width * len(format)
pixels = rawimage.get_data(format, pitch)
```

“format” strings consist of characters that give the byte order of each color component. For example, if rawimage.format is ‘RGBA’, there are four color components: red, green, blue and alpha, in that order. Other common format strings are ‘RGB’, ‘LA’ (luminance, alpha) and ‘I’ (intensity).

The “pitch” of an image is the number of bytes in a row (this may validly be more than the number required to make up the width of the image, it is common to see this for word alignment). If “pitch” is negative the rows of the image are ordered from top to bottom, otherwise they are ordered from bottom to top.

Retrieving data with the format and pitch given in *ImageData.format* and *ImageData.pitch* avoids the need for data conversion (assuming you can make use of the data in this arbitrary format).

Modules

<i>atlas</i>	Group multiple small images into larger textures.
<i>codecs</i>	Collection of image encoders and decoders.

pyglet.image.atlas Group multiple small images into larger textures.

This module is used by *pyglet.resource* to efficiently pack small images into larger textures. *TextureAtlas* maintains one texture; *TextureBin* manages a collection of atlases of a given size.

Example usage:

```
# Load images from disk
car_image = pyglet.image.load('car.png')
boat_image = pyglet.image.load('boat.png')

# Pack these images into one or more textures
bin = TextureBin()
car_texture = bin.add(car_image)
boat_texture = bin.add(boat_image)
```

The result of *TextureBin.add* is a *TextureRegion* containing the image. Once added, an image cannot be removed from a bin (or an atlas); nor can a list of images be obtained from a given bin or atlas – it is the application’s responsibility to keep track of the regions returned by the add methods.

Note: Since pyglet 1.1

<i>Allocator</i>	Rectangular area allocation algorithm.
<i>TextureAtlas</i>	Collection of images within a texture.
<i>TextureBin</i>	Collection of texture atlases.

Classes

pyglet.image.atlas.Allocator

Allocator Class

class Allocator (*width, height*)

Rectangular area allocation algorithm.

Initialise with a given *width* and *height*, then repeatedly call *alloc* to retrieve free regions of the area and protect that area from future allocations.

Allocator uses a fairly simple strips-based algorithm. It performs best when rectangles are allocated in decreasing height order.

pyglet.image.atlas.TextureAtlas

TextureAtlas Class

class TextureAtlas (*width=2048, height=2048*)
Collection of images within a texture.

pyglet.image.atlas.TextureBin

TextureBin Class

class TextureBin (*texture_width=2048, texture_height=2048*)
Collection of texture atlases.

TextureBin maintains a collection of texture atlases, and creates new ones as necessary to accommodate images added to the bin.

AllocatorException The allocator does not have sufficient free space for the requested image size.

Exceptions

pyglet.image.atlas.AllocatorException

AllocatorException

Exception defined in *pyglet.image.atlas*

exception AllocatorException

The allocator does not have sufficient free space for the requested image size.

get_max_texture_size() Query the maximum texture size available

Functions

get_max_texture_size Function Defined in `pyglet.image.atlas`

get_max_texture_size()
Query the maximum texture size available

Variables

division = `_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)`

Defined
<div>Notes</div> <ul style="list-style-type: none"> pyglet

pyglet.image.codecs Collection of image encoders and decoders.

Modules must subclass ImageDecoder and ImageEncoder for each method of decoding/encoding they support.

Modules must also implement the two functions:

```
def get_decoders():
    # Return a list of ImageDecoder instances or []
    return []

def get_encoders():
    # Return a list of ImageEncoder instances or []
    return []
```

<code>bmp</code>	Decoder for BMP files.
<code>dds</code>	DDS texture loader.
<code>gif</code>	Read GIF control data.
<code>png</code>	Encoder and decoder for PNG files, using PyPNG (png.py).
<code>s3tc</code>	Software decoder for S3TC compressed texture (i.e., DDS).

Modules

pyglet.image.codecs.bmp Decoder for BMP files.

Currently supports version 3 and 4 bitmaps with BI_RGB and BI_BITFIELDS encoding. Alpha channel is supported for 32-bit BI_RGB only.

pyglet.image.codecs.dds DDS texture loader.

Reference: <http://msdn2.microsoft.com/en-us/library/bb172993.aspx>

pyglet.image.codecs.gif Read GIF control data.

<http://www.w3.org/Graphics/GIF/spec-gif89a.txt>

`pyglet.image.codecs.png` Encoder and decoder for PNG files, using PyPNG (`png.py`).

`pyglet.image.codecs.s3tc` Software decoder for S3TC compressed texture (i.e., DDS).

http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_s3tc.txt

ImageDecoder

ImageEncoder

Classes

pyglet.image.codecs.ImageDecoder

ImageDecoder Class

`class ImageDecoder`

pyglet.image.codecs.ImageEncoder

ImageEncoder Class

`class ImageEncoder`

ImageDecodeException

ImageEncodeException

Exceptions

pyglet.image.codecs.ImageDecodeException

ImageDecodeException

Exception defined in `pyglet.image.codecs`

exception `ImageDecodeException`

pyglet.image.codecs.ImageEncodeException

ImageEncodeException

Exception defined in `pyglet.image.codecs`

exception `ImageEncodeException`

<code>add_decoders(module)</code>	Add a decoder module.
<code>add_default_image_codecs()</code>	
<code>add_encoders(module)</code>	Add an encoder module.
<code>get_animation_decoders([filename])</code>	Get an ordered list of decoders to attempt.
<code>get_decoders([filename])</code>	Get an ordered list of decoders to attempt.
<code>get_encoders([filename])</code>	Get an ordered list of encoders to attempt.

Functions

`add_decoders` Function Defined in `pyglet.image.codecs`

`add_decoders` (*module*)

Add a decoder module. The module must define `get_decoders`. Once added, the appropriate decoders defined in the codec will be returned by `pyglet.image.codecs.get_decoders`.

`add_default_image_codecs` Function Defined in `pyglet.image.codecs`

`add_default_image_codecs` ()

`add_encoders` Function Defined in `pyglet.image.codecs`

`add_encoders` (*module*)

Add an encoder module. The module must define `get_encoders`. Once added, the appropriate encoders defined in the codec will be returned by `pyglet.image.codecs.get_encoders`.

`get_animation_decoders` Function Defined in `pyglet.image.codecs`

`get_animation_decoders` (*filename=None*)

Get an ordered list of decoders to attempt. `filename` can be used as a hint for the filetype.

`get_decoders` Function Defined in `pyglet.image.codecs`

`get_decoders` (*filename=None*)

Get an ordered list of decoders to attempt. `filename` can be used as a hint for the filetype.

get_encoders Function Defined in `pyglet.image.codecs`

get_encoders (*filename=None*)

Get an ordered list of encoders to attempt. `filename` can be used as a hint for the filetype.

Variables

compat_platform = 'linux'

`str(object='') -> str(str(bytes_or_buffer[, encoding[, errors]]) -> str`

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of `object.__str__()` (if defined) or `repr(object)`. encoding defaults to `sys.getdefaultencoding()`. errors defaults to 'strict'.

Defined

Notes

- `gdkpixbuf2`
- `os`
- `pil`

Classes

<i>AbstractImage</i>	Abstract class representing an image.
<i>AbstractImageSequence</i>	Abstract sequence of images.
<i>Animation</i>	Sequence of images with timing information.
<i>AnimationFrame</i>	A single frame of an animation.
<i>BufferImage</i>	An abstract framebuffer.
<i>BufferImageMask</i>	A single bit of the stencil buffer.
<i>BufferManager</i>	Manages the set of framebuffers for a context.
<i>CheckerImagePattern</i>	Create an image with a tileable checker image.
<i>ColorBufferImage</i>	A color framebuffer.
<i>CompressedImageData</i>	Image representing some compressed data suitable for direct uploading to driver.
<i>DepthBufferImage</i>	The depth buffer.
<i>DepthTexture</i>	A texture with depth samples (typically 24-bit).
<i>ImageData</i>	An image represented as a string of unsigned bytes.
<i>ImageDataRegion</i>	
<i>ImageGrid</i>	An imaginary grid placed over an image allowing easy access to regular regions of that image.
<i>ImagePattern</i>	Abstract image creation class.
<i>SolidColorImagePattern</i>	Creates an image filled with a solid color.
<i>Texture</i>	An image loaded into video memory that can be efficiently drawn to the framebuffer.
<i>Texture3D</i>	A texture with more than one image slice.
<i>TextureGrid</i>	A texture containing a regular grid of texture regions.
<i>TextureRegion</i>	A rectangular region of a texture, presented as if it were a separate texture.
<i>TextureSequence</i>	Interface for a sequence of textures.
<i>TileableTexture</i>	A texture that can be tiled efficiently.
<i>UniformTextureSequence</i>	Interface for a sequence of textures, each with the same dimensions.

pyglet.image.AbstractImage

AbstractImage Class**class AbstractImage** (*width, height*)

Abstract class representing an image.

Variables

- ***width*** – Width of image
- ***height*** – Height of image
- ***anchor_x*** – X coordinate of anchor, relative to left edge of image data
- ***anchor_y*** – Y coordinate of anchor, relative to bottom edge of image data

Attributes:

<i>anchor_x</i>	
<i>anchor_y</i>	
<i>image_data</i>	An <i>ImageData</i> view of this image.
<i>mipmapped_texture</i>	A <i>Texture</i> view of this image.
<i>texture</i>	Get a <i>Texture</i> view of this image.

AttributesAbstractImage.**anchor_x** = 0AbstractImage.**anchor_y** = 0AbstractImage.**image_data**An *ImageData* view of this image.

Changes to the returned instance may or may not be reflected in this image. Read-only.

Warning: Deprecated. Use *get_image_data*.

Type *ImageData*AbstractImage.**mipmapped_texture**A *Texture* view of this image.

The returned *Texture* will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2.
Read-only.

Warning: Deprecated. Use *get_mipmapped_texture*.

Type *Texture*AbstractImage.**texture**Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*

pyglet.image.AbstractImageSequence

AbstractImageSequence Class

class **AbstractImageSequence**

Abstract sequence of images.

The sequence is useful for storing image animations or slices of a volume. For efficient access, use the *texture_sequence* member. The class also implements the sequence interface (`__len__`, `__getitem__`, `__setitem__`).

Attributes:

<i>texture_sequence</i>	Access this image sequence as a texture sequence.
-------------------------	---

Attributes

`AbstractImageSequence.texture_sequence`

Access this image sequence as a texture sequence.

Warning: Deprecated. Use `get_texture_sequence`

Type *TextureSequence*

pyglet.image.Animation

Animation Class

class **Animation** (*frames*)

Sequence of images with timing information.

If no frames of the animation have a duration of `None`, the animation loops continuously; otherwise the animation stops at the first frame with duration of `None`.

Variables **frames** – The frames that make up the animation.

Methods:

<i>from_image_sequence</i> (sequence, period[, loop])	Create an animation from a list of images and a constant framerate.
---	---

Methods

classmethod `Animation.from_image_sequence(sequence, period, loop=True)`

Create an animation from a list of images and a constant framerate.

Parameters

- **sequence** (*list of AbstractImage*) – Images that make up the animation, in sequence.
- **period** (*float*) – Number of seconds to display each image.
- **loop** (*bool*) – If True, the animation will loop continuously.

Return type *Animation*

pyglet.image.AnimationFrame

AnimationFrame Class

class `AnimationFrame(image, duration)`

A single frame of an animation.

pyglet.image.AbstractImage

pyglet.image.BufferImage

**BufferImage Class**

class `BufferImage(x, y, width, height)`

An abstract framebuffer.

Attributes:

<code>anchor_x</code>	
<code>anchor_y</code>	
<code>format</code>	The format string used for image data.
<code>image_data</code>	An <i>ImageData</i> view of this image.
<code>mipmapped_texture</code>	A <i>Texture</i> view of this image.
<code>owner</code>	
<code>texture</code>	Get a <i>Texture</i> view of this image.

Attributes

`BufferImage.format = ''`

The format string used for image data.

`BufferImage.owner = None`

Inherited members

Attributes

`BufferImage.anchor_x = 0`

`BufferImage.anchor_y = 0`

`BufferImage.image_data`

An *ImageData* view of this image.

Changes to the returned instance may or may not be reflected in this image. Read-only.

Warning: Deprecated. Use `get_image_data`.

Type *ImageData*

`BufferImage.mipmapped_texture`

A *Texture* view of this image.

The returned *Texture* will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use `get_mipmapped_texture`.

Type *Texture*

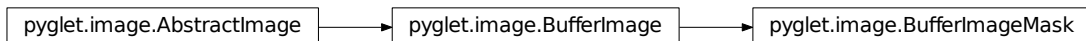
`BufferImage.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*



BufferImageMask Class

class `BufferImageMask` (*x*, *y*, *width*, *height*)

A single bit of the stencil buffer.

Attributes:

`anchor_x`

Continued on next page

Table 2.90 – continued from previous page

<code>anchor_y</code>	
<code>format</code>	
<code>image_data</code>	An <i>ImageData</i> view of this image.
<code>mipmapped_texture</code>	A <i>Texture</i> view of this image.
<code>owner</code>	
<code>texture</code>	Get a <i>Texture</i> view of this image.

Attributes

`BufferImageMask.format = 'L'`

Inherited members

Attributes

`BufferImageMask.anchor_x = 0`

`BufferImageMask.anchor_y = 0`

`BufferImageMask.image_data`
An *ImageData* view of this image.

Changes to the returned instance may or may not be reflected in this image. Read-only.

Warning: Deprecated. Use `get_image_data`.

Type *ImageData*

`BufferImageMask.mipmapped_texture`
A *Texture* view of this image.

The returned *Texture* will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use `get_mipmapped_texture`.

Type *Texture*

`BufferImageMask.owner = None`

`BufferImageMask.texture`
Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*

pyglet.image.BufferManager

BufferManager Class

class BufferManager

Manages the set of framebuffers for a context.

Use *get_buffer_manager* to obtain the instance of this class for the current context.

pyglet.image.ImagePattern

pyglet.image.CheckerImagePattern

CheckerImagePattern Class

class CheckerImagePattern (*color1*=(150, 150, 150, 255), *color2*=(200, 200, 200, 255))

Create an image with a tileable checker image.

pyglet.image.AbstractImage

pyglet.image.BufferImage

pyglet.image.ColorBufferImage

ColorBufferImage Class

class ColorBufferImage (*x*, *y*, *width*, *height*)

A color framebuffer.

This class is used to wrap both the primary color buffer (i.e., the back buffer) or any one of the auxiliary buffers.

Attributes:

<i>anchor_x</i>	
<i>anchor_y</i>	
<i>format</i>	
<i>image_data</i>	An <i>ImageData</i> view of this image.
<i>mipmapped_texture</i>	A <i>Texture</i> view of this image.
<i>owner</i>	
<i>texture</i>	Get a <i>Texture</i> view of this image.

Attributes

`ColorBufferImage.format = 'RGBA'`

Inherited members

Attributes

`ColorBufferImage.anchor_x = 0`

`ColorBufferImage.anchor_y = 0`

`ColorBufferImage.image_data`

An *ImageData* view of this image.

Changes to the returned instance may or may not be reflected in this image. Read-only.

Warning: Deprecated. Use `get_image_data`.

Type *ImageData*

`ColorBufferImage.mipmapped_texture`

A *Texture* view of this image.

The returned *Texture* will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use `get_mipmapped_texture`.

Type *Texture*

`ColorBufferImage.owner = None`

`ColorBufferImage.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*

`pyglet.image.AbstractImage`



`pyglet.image.CompressedImageData`

CompressedImageData Class

class `CompressedImageData` (*width, height, gl_format, data, extension=None, decoder=None*)

Image representing some compressed data suitable for direct uploading to driver.

Attributes:

`anchor_x`

Continued on next page

Table 2.92 – continued from previous page

<code>anchor_y</code>	
<code>image_data</code>	An <i>ImageData</i> view of this image.
<code>mipmapped_texture</code>	A <i>Texture</i> view of this image.
<code>texture</code>	Get a <i>Texture</i> view of this image.

Inherited members

Attributes

`CompressedImageData.anchor_x = 0`

`CompressedImageData.anchor_y = 0`

`CompressedImageData.image_data`

An *ImageData* view of this image.

Changes to the returned instance may or may not be reflected in this image. Read-only.

Warning: Deprecated. Use `get_image_data`.

Type *ImageData*

`CompressedImageData.mipmapped_texture`

A *Texture* view of this image.

The returned *Texture* will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use `get_mipmapped_texture`.

Type *Texture*

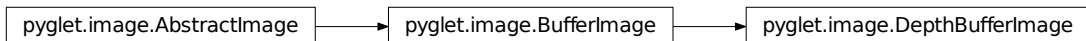
`CompressedImageData.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*



DepthBufferImage Class

class `DepthBufferImage` (*x*, *y*, *width*, *height*)

The depth buffer.

Attributes:

<code>anchor_x</code>	
<code>anchor_y</code>	
<code>format</code>	
<code>image_data</code>	An <i>ImageData</i> view of this image.
<code>mipmapped_texture</code>	A <i>Texture</i> view of this image.
<code>owner</code>	
<code>texture</code>	Get a <i>Texture</i> view of this image.

Attributes

`DepthBufferImage.format = 'L'`

Inherited members

Attributes

`DepthBufferImage.anchor_x = 0`

`DepthBufferImage.anchor_y = 0`

`DepthBufferImage.image_data`

An *ImageData* view of this image.

Changes to the returned instance may or may not be reflected in this image. Read-only.

Warning: Deprecated. Use `get_image_data`.

Type *ImageData*

`DepthBufferImage.mipmapped_texture`

A *Texture* view of this image.

The returned *Texture* will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use `get_mipmapped_texture`.

Type *Texture*

`DepthBufferImage.owner = None`

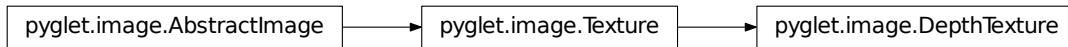
`DepthBufferImage.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*



DepthTexture Class

class `DepthTexture` (*width, height, target, id*)

A texture with depth samples (typically 24-bit).

Methods:

<code>create(width, height[, internalformat, ...])</code>	Create an empty Texture.
<code>create_for_size(target, min_width, min_height)</code>	Create a Texture with dimensions at least <code>min_width</code> , <code>min_height</code> .

Attributes:

<code>anchor_x</code>	
<code>anchor_y</code>	
<code>image_data</code>	An <code>ImageData</code> view of this texture.
<code>images</code>	
<code>level</code>	
<code>mipmapped_texture</code>	A Texture view of this image.
<code>tex_coords</code>	
<code>tex_coords_order</code>	
<code>texture</code>	Get a <i>Texture</i> view of this image.
<code>x</code>	
<code>y</code>	
<code>z</code>	

Inherited members

Methods

`DepthTexture.create` (*width, height, internalformat=6408, rectangle=False, force_rectangle=False, min_filter=9729, mag_filter=9729*)

Create an empty Texture.

If *rectangle* is `False` or the appropriate driver extensions are not available, a larger texture than requested will be created, and a *TextureRegion* corresponding to the requested size will be returned.

Parameters

- **`width`** (*int*) – Width of the texture.
- **`height`** (*int*) – Height of the texture.
- **`internalformat`** (*int*) – GL constant giving the internal format of the texture; for example, `GL_RGBA`.

- **rectangle** (*bool*) – True if a rectangular texture is permitted. See *AbstractImage.get_texture*.
- **force_rectangle** (*bool*) – True if a rectangular texture is required. See *AbstractImage.get_texture*. **Since:** pyglet 1.1.4.
- **min_filter** (*int*) – The minification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`

Return type *Texture*

Note: Since pyglet 1.1

`DepthTexture.create_for_size(target, min_width, min_height, internalformat=None, min_filter=9729, mag_filter=9729)`

Create a Texture with dimensions at least `min_width`, `min_height`. On return, the texture will be bound.

Parameters

- **target** (*int*) – GL constant giving texture target to use, typically `GL_TEXTURE_2D`.
- **min_width** (*int*) – Minimum width of texture (may be increased to create a power of 2).
- **min_height** (*int*) – Minimum height of texture (may be increased to create a power of 2).
- **internalformat** (*int*) – GL constant giving internal format of texture; for example, `GL_RGBA`. If unspecified, the texture will not be initialised (only the texture name will be created on the instance). If specified, the image will be initialised to this format with zero'd data.
- **min_filter** (*int*) – The minification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`

Return type *Texture*

Attributes

`DepthTexture.anchor_x = 0`

`DepthTexture.anchor_y = 0`

`DepthTexture.image_data`

An *ImageData* view of this texture.

Changes to the returned instance will not be reflected in this texture. If the texture is a 3D texture, the first image will be returned. See also *get_image_data*. Read-only.

<p>Warning: Deprecated. Use <i>get_image_data</i>.</p>

Type *ImageData*

`DepthTexture.images = 1`

`DepthTexture.level = 0`

`DepthTexture.mipmapped_texture`

A Texture view of this image.

The returned Texture will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use `get_mipmapped_texture`.

Type *Texture*

`DepthTexture.tex_coords = (0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0)`

`DepthTexture.tex_coords_order = (0, 1, 2, 3)`

`DepthTexture.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

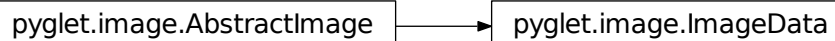
Warning: Deprecated. Use `get_texture`.

Type *Texture*

`DepthTexture.x = 0`

`DepthTexture.y = 0`

`DepthTexture.z = 0`



ImageData Class

class ImageData (*width, height, format, data, pitch=None*)

An image represented as a string of unsigned bytes.

Variables

- ***data*** – Pixel data, encoded according to *format* and *pitch*.
- ***format*** – The format string to use when reading or writing *data*.
- ***pitch*** – Number of bytes per row. Negative values indicate a top-to-bottom arrangement.

Setting the *format* and *pitch* instance variables and reading *data* is deprecated; use `get_data` and `set_data` in new applications. (Reading *format* and *pitch* to obtain the current encoding is not deprecated).

Attributes:

`anchor_x`

Continued on next page

Table 2.96 – continued from previous page

<code>anchor_y</code>	
<code>data</code>	The byte data of the image.
<code>format</code>	Format string of the data.
<code>image_data</code>	An <i>ImageData</i> view of this image.
<code>mipmapped_texture</code>	A <i>Texture</i> view of this image.
<code>texture</code>	Get a <i>Texture</i> view of this image.

Attributes`ImageData.data`

The byte data of the image. Read-write.

Warning: Deprecated. Use `get_data` and `set_data`.**Type** sequence of bytes, or str`ImageData.format`

Format string of the data. Read-write.

Type str**Inherited members****Attributes**`ImageData.anchor_x = 0``ImageData.anchor_y = 0``ImageData.image_data`An *ImageData* view of this image.

Changes to the returned instance may or may not be reflected in this image. Read-only.

Warning: Deprecated. Use `get_image_data`.**Type** *ImageData*`ImageData.mipmapped_texture`A *Texture* view of this image.The returned *Texture* will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.**Warning:** Deprecated. Use `get_mipmapped_texture`.**Type** *Texture*`ImageData.texture`Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*



ImageDataRegion Class

class ImageDataRegion (*x, y, width, height, image_data*)

Attributes:

<code>anchor_x</code>	
<code>anchor_y</code>	
<code>data</code>	
<code>format</code>	Format string of the data.
<code>image_data</code>	An <i>ImageData</i> view of this image.
<code>mipmapped_texture</code>	A Texture view of this image.
<code>texture</code>	Get a <i>Texture</i> view of this image.

Attributes

`ImageDataRegion.data`

Inherited members

Attributes

`ImageDataRegion.anchor_x = 0`

`ImageDataRegion.anchor_y = 0`

`ImageDataRegion.format`
Format string of the data. Read-write.

Type `str`

`ImageDataRegion.image_data`
An *ImageData* view of this image.

Changes to the returned instance may or may not be reflected in this image. Read-only.

Warning: Deprecated. Use `get_image_data`.

Type *ImageData*

`ImageDataRegion.mipmapped_texture`
A Texture view of this image.

The returned Texture will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use `get_mipmapped_texture`.

Type *Texture*

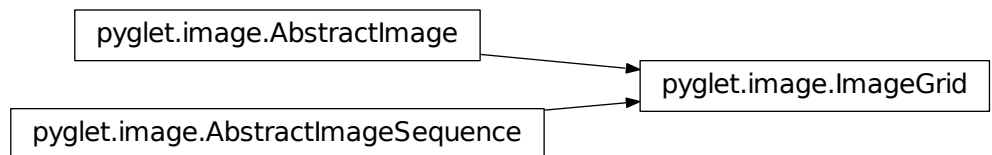
`ImageDataRegion.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*



ImageGrid Class

class ImageGrid(*image, rows, columns, item_width=None, item_height=None, row_padding=0, column_padding=0*)

An imaginary grid placed over an image allowing easy access to regular regions of that image.

The grid can be accessed either as a complete image, or as a sequence of images. The most useful applications are to access the grid as a *TextureGrid*:

```
image_grid = ImageGrid(...)
texture_grid = image_grid.get_texture_sequence()
```

or as a *Texture3D*:

```
image_grid = ImageGrid(...)
texture_3d = Texture3D.create_for_image_grid(image_grid)
```

Attributes:

<code>anchor_x</code>	
<code>anchor_y</code>	
<code>image_data</code>	An <i>ImageData</i> view of this image.
<code>mipmapped_texture</code>	A <i>Texture</i> view of this image.
<code>texture</code>	Get a <i>Texture</i> view of this image.
<code>texture_sequence</code>	Access this image sequence as a texture sequence.

Inherited members

Attributes

`ImageGrid.anchor_x = 0`

`ImageGrid.anchor_y = 0`

`ImageGrid.image_data`

An *ImageData* view of this image.

Changes to the returned instance may or may not be reflected in this image. Read-only.

Warning: Deprecated. Use *get_image_data*.

Type *ImageData*

`ImageGrid.mipmapped_texture`

A *Texture* view of this image.

The returned *Texture* will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use *get_mipmapped_texture*.

Type *Texture*

`ImageGrid.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use *get_texture*.

Type *Texture*

`ImageGrid.texture_sequence`

Access this image sequence as a texture sequence.

Warning: Deprecated. Use *get_texture_sequence*

Type *TextureSequence*

pyglet.image.ImagePattern

ImagePattern Class

class ImagePattern

Abstract image creation class.

pyglet.image.ImagePattern



pyglet.image.SolidColorImagePattern

SolidColorImagePattern Class

class SolidColorImagePattern (*color=(0, 0, 0, 0)*)
Creates an image filled with a solid color.

pyglet.image.AbstractImage



pyglet.image.Texture

Texture Class

class Texture (*width, height, target, id*)
An image loaded into video memory that can be efficiently drawn to the framebuffer.
Typically you will get an instance of Texture by accessing the *texture* member of any other AbstractImage.

Variables

- **region_class** – Class to use when constructing regions of this texture.
- **tex_coords** – 12-tuple of float, named (u1, v1, r1, u2, v2, r2, ...). u, v, r give the 3D texture coordinates for vertices 1-4. The vertices are specified in the order bottom-left, bottom-right, top-right and top-left.
- **target** – The GL texture target (e.g., GL_TEXTURE_2D).
- **level** – The mipmap level of this texture.

Methods:

<code>create</code> (width, height[, internalformat, ...])	Create an empty Texture.
<code>create_for_size</code> (target, min_width, min_height)	Create a Texture with dimensions at least min_width, min_height.

Attributes:

<code>anchor_x</code>	
<code>anchor_y</code>	
<code>image_data</code>	An ImageData view of this texture.
<code>images</code>	
<code>level</code>	
<code>mipmapped_texture</code>	A Texture view of this image.
<code>tex_coords</code>	
<code>tex_coords_order</code>	

Continued on next page

Table 2.100 – continued from previous page

texture	Get a <i>Texture</i> view of this image.
<i>x</i>	
<i>y</i>	
<i>z</i>	

Methods

classmethod `Texture.create`(*width*, *height*, *internalformat*=6408, *rectangle*=False, *force_rectangle*=False, *min_filter*=9729, *mag_filter*=9729)

Create an empty Texture.

If *rectangle* is False or the appropriate driver extensions are not available, a larger texture than requested will be created, and a *TextureRegion* corresponding to the requested size will be returned.

Parameters

- **width** (*int*) – Width of the texture.
- **height** (*int*) – Height of the texture.
- **internalformat** (*int*) – GL constant giving the internal format of the texture; for example, GL_RGBA.
- **rectangle** (*bool*) – True if a rectangular texture is permitted. See *AbstractImage.get_texture*.
- **force_rectangle** (*bool*) – True if a rectangular texture is required. See *AbstractImage.get_texture*. **Since:** pyglet 1.1.4.
- **min_filter** (*int*) – The minification filter used for this texture, commonly GL_LINEAR or GL_NEAREST.
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly GL_LINEAR or GL_NEAREST.

Return type *Texture*

Note: Since pyglet 1.1

classmethod `Texture.create_for_size`(*target*, *min_width*, *min_height*, *internalformat*=None, *min_filter*=9729, *mag_filter*=9729)

Create a Texture with dimensions at least *min_width*, *min_height*. On return, the texture will be bound.

Parameters

- **target** (*int*) – GL constant giving texture target to use, typically GL_TEXTURE_2D.
- **min_width** (*int*) – Minimum width of texture (may be increased to create a power of 2).
- **min_height** (*int*) – Minimum height of texture (may be increased to create a power of 2).
- **internalformat** (*int*) – GL constant giving internal format of texture; for example, GL_RGBA. If unspecified, the texture will not be initialised (only the texture name will be created on the instance). If specified, the image will be initialised to this format with zero'd data.

- **min_filter** (*int*) – The minification filter used for this texture, commonly GL_LINEAR or GL_NEAREST
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly GL_LINEAR or GL_NEAREST

Return type *Texture*

Attributes

Texture.image_data

An ImageData view of this texture.

Changes to the returned instance will not be reflected in this texture. If the texture is a 3D texture, the first image will be returned. See also *get_image_data*. Read-only.

Warning: Deprecated. Use *get_image_data*.

Type *ImageData*

Texture.images = 1

Texture.level = 0

Texture.tex_coords = (0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0)

Texture.tex_coords_order = (0, 1, 2, 3)

Texture.x = 0

Texture.y = 0

Texture.z = 0

Inherited members

Attributes

Texture.anchor_x = 0

Texture.anchor_y = 0

Texture.mipmapped_texture

A Texture view of this image.

The returned Texture will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use *get_mipmapped_texture*.

Type *Texture*

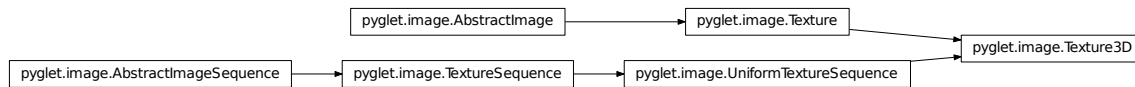
Texture.texture

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use *get_texture*.

Type *Texture*



Texture3D Class

class Texture3D (*width, height, target, id*)

A texture with more than one image slice.

Use *create_for_images* or *create_for_image_grid* classmethod to construct.

Methods:

<code>create(width, height[, internalformat, ...])</code>	Create an empty Texture.
<code>create_for_image_grid(grid[, internalformat])</code>	
<code>create_for_images(images[, internalformat])</code>	
<code>create_for_size(target, min_width, min_height)</code>	Create a Texture with dimensions at least min_width, min_height.

Attributes:

<code>anchor_x</code>	
<code>anchor_y</code>	
<code>image_data</code>	An ImageData view of this texture.
<code>images</code>	
<code>item_height</code>	
<code>item_width</code>	
<code>items</code>	
<code>level</code>	
<code>mipmapped_texture</code>	A Texture view of this image.
<code>tex_coords</code>	
<code>tex_coords_order</code>	
<code>texture</code>	Get a <i>Texture</i> view of this image.
<code>texture_sequence</code>	Access this image sequence as a texture sequence.
<code>x</code>	
<code>y</code>	
<code>z</code>	

Methods

classmethod Texture3D.**create_for_image_grid** (*grid, internalformat=6408*)

classmethod Texture3D.**create_for_images** (*images, internalformat=6408*)

Attributes

Texture3D.**item_height** = 0

Texture3D.**item_width** = 0

Texture3D.**items** = ()

Inherited members

Methods

`Texture3D.create` (*width*, *height*, *internalformat*=6408, *rectangle*=False, *force_rectangle*=False, *min_filter*=9729, *mag_filter*=9729)

Create an empty Texture.

If *rectangle* is False or the appropriate driver extensions are not available, a larger texture than requested will be created, and a *TextureRegion* corresponding to the requested size will be returned.

Parameters

- **width** (*int*) – Width of the texture.
- **height** (*int*) – Height of the texture.
- **internalformat** (*int*) – GL constant giving the internal format of the texture; for example, GL_RGBA.
- **rectangle** (*bool*) – True if a rectangular texture is permitted. See *AbstractImage.get_texture*.
- **force_rectangle** (*bool*) – True if a rectangular texture is required. See *AbstractImage.get_texture*. **Since:** pyglet 1.1.4.
- **min_filter** (*int*) – The minification filter used for this texture, commonly GL_LINEAR or GL_NEAREST
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly GL_LINEAR or GL_NEAREST

Return type *Texture*

Note: Since pyglet 1.1

`Texture3D.create_for_size` (*target*, *min_width*, *min_height*, *internalformat*=None, *min_filter*=9729, *mag_filter*=9729)

Create a Texture with dimensions at least *min_width*, *min_height*. On return, the texture will be bound.

Parameters

- **target** (*int*) – GL constant giving texture target to use, typically GL_TEXTURE_2D.
- **min_width** (*int*) – Minimum width of texture (may be increased to create a power of 2).
- **min_height** (*int*) – Minimum height of texture (may be increased to create a power of 2).
- **internalformat** (*int*) – GL constant giving internal format of texture; for example, GL_RGBA. If unspecified, the texture will not be initialised (only the texture name will be created on the instance). If specified, the image will be initialised to this format with zero'd data.
- **min_filter** (*int*) – The minification filter used for this texture, commonly GL_LINEAR or GL_NEAREST

- **mag_filter** (*int*) – The magnification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`

Return type *Texture*

Attributes

`Texture3D.anchor_x = 0`

`Texture3D.anchor_y = 0`

`Texture3D.image_data`

An *ImageData* view of this texture.

Changes to the returned instance will not be reflected in this texture. If the texture is a 3D texture, the first image will be returned. See also *get_image_data*. Read-only.

Warning: Deprecated. Use *get_image_data*.

Type *ImageData*

`Texture3D.images = 1`

`Texture3D.level = 0`

`Texture3D.mipmapped_texture`

A *Texture* view of this image.

The returned *Texture* will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use *get_mipmapped_texture*.

Type *Texture*

`Texture3D.tex_coords = (0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0)`

`Texture3D.tex_coords_order = (0, 1, 2, 3)`

`Texture3D.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use *get_texture*.

Type *Texture*

`Texture3D.texture_sequence`

Access this image sequence as a texture sequence.

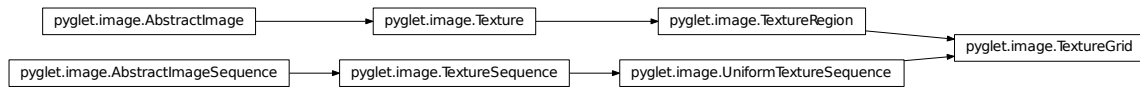
Warning: Deprecated. Use *get_texture_sequence*

Type *TextureSequence*

`Texture3D.x = 0`

`Texture3D.y = 0`

`Texture3D.z = 0`



TextureGrid Class

class TextureGrid(*grid*)

A texture containing a regular grid of texture regions.

To construct, create an *ImageGrid* first:

```
image_grid = ImageGrid(...)
texture_grid = TextureGrid(image_grid)
```

The texture grid can be accessed as a single texture, or as a sequence of *TextureRegion*. When accessing as a sequence, you can specify integer indexes, in which the images are arranged in rows from the bottom-left to the top-right:

```
# assume the texture_grid is 3x3:
current_texture = texture_grid[3] # get the middle-left image
```

You can also specify tuples in the sequence methods, which are addressed as row, column:

```
# equivalent to the previous example:
current_texture = texture_grid[1, 0]
```

When using tuples in a slice, the returned sequence is over the rectangular region defined by the slice:

```
# returns center, center-right, center-top, top-right images in that
# order:
images = texture_grid[(1,1):]
# equivalent to
images = texture_grid[(1,1):(3,3)]
```

Methods:

<code>create(width, height[, internalformat, ...])</code>	Create an empty Texture.
<code>create_for_size(target, min_width, min_height)</code>	Create a Texture with dimensions at least min_width, min_height.

Attributes:

<code>anchor_x</code>	
<code>anchor_y</code>	
<code>columns</code>	
<code>image_data</code>	An ImageData view of this texture.
<code>images</code>	
<code>item_height</code>	
<code>item_width</code>	
<code>items</code>	
<code>level</code>	
<code>mipmapped_texture</code>	A Texture view of this image.
<code>rows</code>	

Continued on next page

Table 2.104 – continued from previous page

<code>tex_coords</code>	
<code>tex_coords_order</code>	
<code>texture</code>	Get a <i>Texture</i> view of this image.
<code>texture_sequence</code>	Access this image sequence as a texture sequence.
<code>x</code>	
<code>y</code>	
<code>z</code>	

Attributes

```
TextureGrid.columns = 1
TextureGrid.item_height = 0
TextureGrid.item_width = 0
TextureGrid.items = ()
TextureGrid.rows = 1
```

Inherited members

Methods

```
TextureGrid.create(width, height, internalformat=6408, rectangle=False,
                   force_rectangle=False, min_filter=9729, mag_filter=9729)
```

Create an empty Texture.

If *rectangle* is *False* or the appropriate driver extensions are not available, a larger texture than requested will be created, and a *TextureRegion* corresponding to the requested size will be returned.

Parameters

- **width** (*int*) – Width of the texture.
- **height** (*int*) – Height of the texture.
- **internalformat** (*int*) – GL constant giving the internal format of the texture; for example, `GL_RGBA`.
- **rectangle** (*bool*) – True if a rectangular texture is permitted. See *AbstractImage.get_texture*.
- **force_rectangle** (*bool*) – True if a rectangular texture is required. See *AbstractImage.get_texture*. **Since:** pyglet 1.1.4.
- **min_filter** (*int*) – The minification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`

Return type *Texture*

Note: Since pyglet 1.1

`TextureGrid.create_for_size(target, min_width, min_height, internalformat=None, min_filter=9729, mag_filter=9729)`

Create a Texture with dimensions at least `min_width`, `min_height`. On return, the texture will be bound.

Parameters

- **target** (*int*) – GL constant giving texture target to use, typically `GL_TEXTURE_2D`.
- **min_width** (*int*) – Minimum width of texture (may be increased to create a power of 2).
- **min_height** (*int*) – Minimum height of texture (may be increased to create a power of 2).
- **internalformat** (*int*) – GL constant giving internal format of texture; for example, `GL_RGBA`. If unspecified, the texture will not be initialised (only the texture name will be created on the instance). If specified, the image will be initialised to this format with zero'd data.
- **min_filter** (*int*) – The minification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`

Return type *Texture*

Attributes

`TextureGrid.anchor_x = 0`

`TextureGrid.anchor_y = 0`

`TextureGrid.image_data`

An `ImageData` view of this texture.

Changes to the returned instance will not be reflected in this texture. If the texture is a 3D texture, the first image will be returned. See also `get_image_data`. Read-only.

Warning: Deprecated. Use `get_image_data`.

Type *ImageData*

`TextureGrid.images = 1`

`TextureGrid.level = 0`

`TextureGrid.mipmapped_texture`

A Texture view of this image.

The returned Texture will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use `get_mipmapped_texture`.

Type *Texture*

`TextureGrid.tex_coords = (0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0)`

`TextureGrid.tex_coords_order = (0, 1, 2, 3)`

`TextureGrid.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*

`TextureGrid.texture_sequence`

Access this image sequence as a texture sequence.

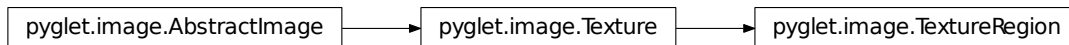
Warning: Deprecated. Use `get_texture_sequence`

Type *TextureSequence*

`TextureGrid.x = 0`

`TextureGrid.y = 0`

`TextureGrid.z = 0`



TextureRegion Class

class `TextureRegion` (*x, y, z, width, height, owner*)

A rectangular region of a texture, presented as if it were a separate texture.

Methods:

<code>create(width, height[, internalformat, ...])</code>	Create an empty <i>Texture</i> .
<code>create_for_size(target, min_width, min_height)</code>	Create a <i>Texture</i> with dimensions at least <code>min_width</code> , <code>min_height</code> .

Attributes:

<code>anchor_x</code>	
<code>anchor_y</code>	
<code>image_data</code>	An <i>ImageData</i> view of this texture.
<code>images</code>	
<code>level</code>	
<code>mipmapped_texture</code>	A <i>Texture</i> view of this image.
<code>tex_coords</code>	
<code>tex_coords_order</code>	
<code>texture</code>	Get a <i>Texture</i> view of this image.
<code>x</code>	
<code>y</code>	
<code>z</code>	

Inherited members

Methods

`TextureRegion.create` (*width*, *height*, *internalformat*=6408, *rectangle*=False, *force_rectangle*=False, *min_filter*=9729, *mag_filter*=9729)

Create an empty Texture.

If *rectangle* is False or the appropriate driver extensions are not available, a larger texture than requested will be created, and a *TextureRegion* corresponding to the requested size will be returned.

Parameters

- **width** (*int*) – Width of the texture.
- **height** (*int*) – Height of the texture.
- **internalformat** (*int*) – GL constant giving the internal format of the texture; for example, GL_RGBA.
- **rectangle** (*bool*) – True if a rectangular texture is permitted. See *AbstractImage.get_texture*.
- **force_rectangle** (*bool*) – True if a rectangular texture is required. See *AbstractImage.get_texture*. **Since:** pyglet 1.1.4.
- **min_filter** (*int*) – The minification filter used for this texture, commonly GL_LINEAR or GL_NEAREST
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly GL_LINEAR or GL_NEAREST

Return type *Texture*

Note: Since pyglet 1.1

`TextureRegion.create_for_size` (*target*, *min_width*, *min_height*, *internalformat*=None, *min_filter*=9729, *mag_filter*=9729)

Create a Texture with dimensions at least *min_width*, *min_height*. On return, the texture will be bound.

Parameters

- **target** (*int*) – GL constant giving texture target to use, typically GL_TEXTURE_2D.
- **min_width** (*int*) – Minimum width of texture (may be increased to create a power of 2).
- **min_height** (*int*) – Minimum height of texture (may be increased to create a power of 2).
- **internalformat** (*int*) – GL constant giving internal format of texture; for example, GL_RGBA. If unspecified, the texture will not be initialised (only the texture name will be created on the instance). If specified, the image will be initialised to this format with zero'd data.
- **min_filter** (*int*) – The minification filter used for this texture, commonly GL_LINEAR or GL_NEAREST

- **mag_filter** (*int*) – The magnification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`

Return type *Texture*

Attributes

`TextureRegion.anchor_x = 0`

`TextureRegion.anchor_y = 0`

`TextureRegion.image_data`

An *ImageData* view of this texture.

Changes to the returned instance will not be reflected in this texture. If the texture is a 3D texture, the first image will be returned. See also *get_image_data*. Read-only.

Warning: Deprecated. Use *get_image_data*.

Type *ImageData*

`TextureRegion.images = 1`

`TextureRegion.level = 0`

`TextureRegion.mipmapped_texture`

A *Texture* view of this image.

The returned *Texture* will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use *get_mipmapped_texture*.

Type *Texture*

`TextureRegion.tex_coords = (0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0)`

`TextureRegion.tex_coords_order = (0, 1, 2, 3)`

`TextureRegion.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

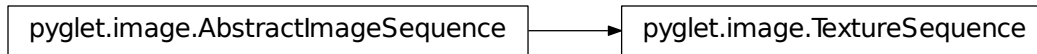
Warning: Deprecated. Use *get_texture*.

Type *Texture*

`TextureRegion.x = 0`

`TextureRegion.y = 0`

`TextureRegion.z = 0`



TextureSequence Class

class TextureSequence

Interface for a sequence of textures.

Typical implementations store multiple *TextureRegion* s within one *Texture* so as to minimise state changes.

Attributes:

<code>texture_sequence</code>	Access this image sequence as a texture sequence.
-------------------------------	---

Inherited members

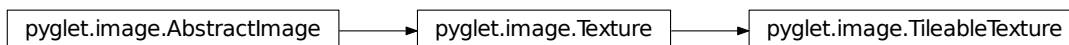
Attributes

`TextureSequence.texture_sequence`

Access this image sequence as a texture sequence.

Warning: Deprecated. Use `get_texture_sequence`

Type *TextureSequence*



TileableTexture Class

class TileableTexture (*width, height, target, id*)

A texture that can be tiled efficiently.

Use `create_for_image` classmethod to construct.

Methods:

<code>create(width, height[, internalformat, ...])</code>	Create an empty Texture.
<code>create_for_image(image)</code>	
<code>create_for_size(target, min_width, min_height)</code>	Create a Texture with dimensions at least min_width, min_height.

Attributes:

anchor_x	
anchor_y	
image_data	An ImageData view of this texture.
images	
level	
mipmapped_texture	A Texture view of this image.
tex_coords	
tex_coords_order	
texture	Get a <i>Texture</i> view of this image.
x	
y	
z	

Methods

classmethod `TileableTexture.create_for_image(image)`

Inherited members

Methods

`TileableTexture.create(width, height, internalformat=6408, rectangle=False, force_rectangle=False, min_filter=9729, mag_filter=9729)`

Create an empty Texture.

If *rectangle* is `False` or the appropriate driver extensions are not available, a larger texture than requested will be created, and a *TextureRegion* corresponding to the requested size will be returned.

Parameters

- **width** (*int*) – Width of the texture.
- **height** (*int*) – Height of the texture.
- **internalformat** (*int*) – GL constant giving the internal format of the texture; for example, `GL_RGBA`.
- **rectangle** (*bool*) – True if a rectangular texture is permitted. See *AbstractImage.get_texture*.
- **force_rectangle** (*bool*) – True if a rectangular texture is required. See *AbstractImage.get_texture*. **Since:** pyglet 1.1.4.
- **min_filter** (*int*) – The minification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`

Return type *Texture*

Note: Since pyglet 1.1

`TileableTexture.create_for_size(target, min_width, min_height, internalformat=None, min_filter=9729, mag_filter=9729)`

Create a Texture with dimensions at least `min_width`, `min_height`. On return, the texture will be bound.

Parameters

- **target** (*int*) – GL constant giving texture target to use, typically `GL_TEXTURE_2D`.
- **min_width** (*int*) – Minimum width of texture (may be increased to create a power of 2).
- **min_height** (*int*) – Minimum height of texture (may be increased to create a power of 2).
- **internalformat** (*int*) – GL constant giving internal format of texture; for example, `GL_RGBA`. If unspecified, the texture will not be initialised (only the texture name will be created on the instance). If specified, the image will be initialised to this format with zero'd data.
- **min_filter** (*int*) – The minification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`
- **mag_filter** (*int*) – The magnification filter used for this texture, commonly `GL_LINEAR` or `GL_NEAREST`

Return type *Texture*

Attributes

`TileableTexture.anchor_x = 0`

`TileableTexture.anchor_y = 0`

`TileableTexture.image_data`

An `ImageData` view of this texture.

Changes to the returned instance will not be reflected in this texture. If the texture is a 3D texture, the first image will be returned. See also `get_image_data`. Read-only.

Warning: Deprecated. Use `get_image_data`.

Type *ImageData*

`TileableTexture.images = 1`

`TileableTexture.level = 0`

`TileableTexture.mipmapped_texture`

A Texture view of this image.

The returned Texture will have mipmaps filled in for all levels. Requires that image dimensions be powers of 2. Read-only.

Warning: Deprecated. Use `get_mipmapped_texture`.

Type *Texture*

`TileableTexture.tex_coords = (0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0)`

`TileableTexture.tex_coords_order = (0, 1, 2, 3)`

`TileableTexture.texture`

Get a *Texture* view of this image.

Changes to the returned instance may or may not be reflected in this image.

Warning: Deprecated. Use `get_texture`.

Type *Texture*

`TileableTexture.x = 0`

`TileableTexture.y = 0`

`TileableTexture.z = 0`



UniformTextureSequence Class

class `UniformTextureSequence`

Interface for a sequence of textures, each with the same dimensions.

Variables

- *item_width* – Width of each texture in the sequence.
- *item_height* – Height of each texture in the sequence.

Attributes:

<i>item_height</i>	
<i>item_width</i>	
<i>texture_sequence</i>	Access this image sequence as a texture sequence.

Attributes

`UniformTextureSequence.item_height`

`UniformTextureSequence.item_width`

Inherited members

Attributes

`UniformTextureSequence.texture_sequence`

Access this image sequence as a texture sequence.

Warning: Deprecated. Use `get_texture_sequence`

Type *TextureSequence*

Exceptions

ImageException

pyglet.image.ImageException

ImageException

Exception defined in `pyglet.image`

exception `ImageException`

Functions

<code>color_as_bytes(color)</code>	
<code>create(width, height[, pattern])</code>	Create an image optionally filled with the given pattern.
<code>get_buffer_manager()</code>	Get the buffer manager for the current OpenGL context.
<code>load(filename[, file, decoder])</code>	Load an image from a file.
<code>load_animation(filename[, file, decoder])</code>	Load an animation from a file.

`color_as_bytes` Function Defined in `pyglet.image`

`color_as_bytes` (*color*)

`create` Function Defined in `pyglet.image`

`create` (*width, height, pattern=None*)

Create an image optionally filled with the given pattern.

Note You can make no assumptions about the return type; usually it will be `ImageData` or `CompressedImageData`, but patterns are free to return any subclass of `AbstractImage`.

Parameters

- **`width`** (*int*) – Width of image to create
- **`height`** (*int*) – Height of image to create
- **`pattern`** (*ImagePattern* or *None*) – Pattern to fill image with. If unspecified, the image will initially be transparent.

Return type *AbstractImage*

`get_buffer_manager` Function Defined in `pyglet.image`

`get_buffer_manager` ()

Get the buffer manager for the current OpenGL context.

Return type *BufferManager*

load Function Defined in `pyglet.image`

load (*filename*, *file=None*, *decoder=None*)

Load an image from a file.

Note You can make no assumptions about the return type; usually it will be `ImageData` or `CompressedImageData`, but decoders are free to return any subclass of `AbstractImage`.

Parameters

- **filename** (*str*) – Used to guess the image format, and to load the file if *file* is unspecified.
- **file** (*file-like object or None*) – Source of image data in any supported format.
- **decoder** (*ImageDecoder or None*) – If unspecified, all decoders that are registered for the filename extension are tried. If none succeed, the exception from the first decoder is raised.

Return type `AbstractImage`

load_animation Function Defined in `pyglet.image`

load_animation (*filename*, *file=None*, *decoder=None*)

Load an animation from a file.

Currently, the only supported format is GIF.

Parameters

- **filename** (*str*) – Used to guess the animation format, and to load the file if *file* is unspecified.
- **file** (*file-like object or None*) – File object containing the animation stream.
- **decoder** (*ImageDecoder or None*) – If unspecified, all decoders that are registered for the filename extension are tried. If none succeed, the exception from the first decoder is raised.

Return type `Animation`

Variables

absolute_import = `_Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)`

compat_platform = `'linux'`

`str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str`

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of `object.__str__()` (if defined) or `repr(object)`. encoding defaults to `sys.getdefaultencoding()`. errors defaults to `'strict'`.

division = `_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)`

print_function = `_Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)`

Notes

Defined

- event
- gl
- glext_arb
- glu
- graphics
- key
- lib
- lib_glx
- mouse
- pyglet
- re
- sys
- util
- warnings
- weakref

pyglet.info

Get environment information useful for debugging.

Intended usage is to create a file for bug reports, e.g.:

```
python -m pyglet.info > info.txt
```

Functions

<code>dump()</code>	Dump all information to stdout.
<code>dump_al()</code>	Dump OpenAL info.
<code>dump_avbin()</code>	Dump AVbin info.
<code>dump_gl([context])</code>	Dump GL info.
<code>dump_glu()</code>	Dump GLU info.
<code>dump_glx()</code>	Dump GLX info.
<code>dump_media()</code>	Dump pyglet.media info.
<code>dump_pyglet()</code>	Dump pyglet version and options.
<code>dump_python()</code>	Dump Python version and environment to stdout.
<code>dump_window()</code>	Dump display, window, screen and default config info.
<code>dump_wintab()</code>	Dump WinTab info.

dump Function Defined in `pyglet.info`

dump()
Dump all information to stdout.

dump_al Function Defined in `pyglet.info`

dump_al()
Dump OpenAL info.

dump_avbin Function Defined in *pyglet.info*

dump_avbin()
Dump AVbin info.

dump_gl Function Defined in *pyglet.info*

dump_gl(context=None)
Dump GL info.

dump_glu Function Defined in *pyglet.info*

dump_glu()
Dump GLU info.

dump_glx Function Defined in *pyglet.info*

dump_glx()
Dump GLX info.

dump_media Function Defined in *pyglet.info*

dump_media()
Dump pyglet.media info.

dump_pyglet Function Defined in *pyglet.info*

dump_pyglet()
Dump pyglet version and options.

dump_python Function Defined in *pyglet.info*

dump_python()
Dump Python version and environment to stdout.

dump_window Function Defined in *pyglet.info*

dump_window()
Dump display, window, screen and default config info.

dump_wintab Function Defined in *pyglet.info*

dump_wintab()
Dump WinTab info.

Variables

print_function = `_Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)`

pyglet.input

Joystick, tablet and USB HID device support.

This module provides a unified interface to almost any input device, besides the regular mouse and keyboard support provided by *Window*. At the lowest level, *get_devices* can be used to retrieve a list of all supported devices, including joysticks, tablets, space controllers, wheels, pedals, remote controls, keyboards and mice. The set of returned devices varies greatly depending on the operating system (and, of course, what's plugged in).

At this level pyglet does not try to interpret *what* a particular device is, merely what controls it provides. A *Control* can be either a button, whose value is either `True` or `False`, or a relative or absolute-valued axis, whose value is a float. Sometimes the name of a control can be provided (for example, `x`, representing the horizontal axis of a joystick), but often not. In these cases the device API may still be useful – the user will have to be asked to press each button in turn or move each axis separately to identify them.

Higher-level interfaces are provided for joysticks, tablets and the Apple remote control. These devices can usually be identified by pyglet positively, and a base level of functionality for each one provided through a common interface.

To use an input device:

1. Call *get_devices*, *get_apple_remote* or *get_joysticks* to retrieve and identify the device.
2. For low-level devices (retrieved by *get_devices*), query the devices list of controls and determine which ones you are interested in. For high-level interfaces the set of controls is provided by the interface.
3. Optionally attach event handlers to controls on the device.
4. Call *Device.open* to begin receiving events on the device. You can begin querying the control values after this time; they will be updated asynchronously.
5. Call *Device.close* when you are finished with the device (not needed if your application quits at this time).

To use a tablet, follow the procedure above using *get_tablets*, but note that no control list is available; instead, calling *Tablet.open* returns a *TabletCanvas* onto which you should set your event handlers.

Note: Since pyglet 1.2

Modules

<i>base</i>	Interface classes for <i>pyglet.input</i> .
<i>evdev_constants</i>	Event constants from <code>/usr/include/linux/input.h</code>

pyglet.input.base Interface classes for *pyglet.input*.

Note: Since pyglet 1.2

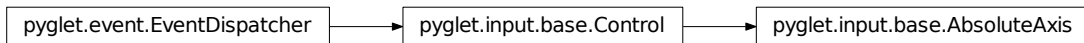
<i>AbsoluteAxis</i>	An axis whose value represents a physical measurement from the device.
<i>AppleRemote</i>	High-level interface for Apple remote control.
<i>Button</i>	A control whose value is boolean.
<i>Control</i>	Single value input provided by a device.
<i>Device</i>	Input device.

Continued on next page

Table 2.115 – continued from previous page

<i>Joystick</i>	High-level interface for joystick-like devices.
<i>RelativeAxis</i>	An axis whose value represents a relative change from the previous value.
<i>Tablet</i>	High-level interface to tablet devices.
<i>TabletCanvas</i>	Event dispatcher for tablets.
<i>TabletCursor</i>	A distinct cursor used on a tablet.

Classes



AbsoluteAxis Class

class *AbsoluteAxis* (*name*, *min*, *max*, *raw_name*=None)

An axis whose value represents a physical measurement from the device.

The value is advertised to range over `min` and `max`.

Variables

- **`min`** – Minimum advertised value.
- **`max`** – Maximum advertised value.

Methods:

Attributes:

<i>HAT</i>	Name of the hat (POV) control, when a single control enumerates all of the hat’s positions.
<i>HAT_X</i>	Name of the hat’s (POV’s) horizontal control, when the hat position is described by two orthogonal controls.
<i>HAT_Y</i>	Name of the hat’s (POV’s) vertical control, when the hat position is described by two orthogonal controls.
<i>RX</i>	Name of the rotational-X axis control
<i>RY</i>	Name of the rotational-Y axis control
<i>RZ</i>	Name of the rotational-Z axis control
<i>X</i>	Name of the horizontal axis control
<i>Y</i>	Name of the vertical axis control
<i>Z</i>	Name of the Z axis control.
<code>event_types</code>	
<code>value</code>	Current value of the control.

Attributes

`AbsoluteAxis.HAT` = ‘hat’

Name of the hat (POV) control, when a single control enumerates all of the hat’s positions.

`AbsoluteAxis.HAT_X` = ‘hat_x’

Name of the hat’s (POV’s) horizontal control, when the hat position is described by two orthogonal controls.

`AbsoluteAxis.HAT_Y = 'hat_y'`

Name of the hat's (POV's) vertical control, when the hat position is described by two orthogonal controls.

`AbsoluteAxis.RX = 'rx'`

Name of the rotational-X axis control

`AbsoluteAxis.RY = 'ry'`

Name of the rotational-Y axis control

`AbsoluteAxis.RZ = 'rz'`

Name of the rotational-Z axis control

`AbsoluteAxis.X = 'x'`

Name of the horizontal axis control

`AbsoluteAxis.Y = 'y'`

Name of the vertical axis control

`AbsoluteAxis.Z = 'z'`

Name of the Z axis control.

Inherited members

Methods

`AbsoluteAxis.register_event_type(name)`

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters `name` (*str*) – Name of the event to register.

Attributes

`AbsoluteAxis.event_types = ['on_change', 'on_press', 'on_release']`

`AbsoluteAxis.value`

Current value of the control.

The range of the value is device-dependent; for absolute controls the range is given by `min` and `max` (however the value may exceed this range); for relative controls the range is undefined.

Type float

pyglet.event.EventDispatcher

pyglet.input.base.AppleRemote

AppleRemote Class

class **AppleRemote** (*device*)

High-level interface for Apple remote control.

This interface provides access to the 6 button controls on the remote. Pressing and holding certain buttons on the remote is interpreted as a separate control.

Variables

- **device** – The underlying device used by this interface.
- **left_control** – Button control for the left (prev) button.
- **left_hold_control** – Button control for holding the left button (rewind).
- **right_control** – Button control for the right (next) button.
- **right_hold_control** – Button control for holding the right button (fast forward).
- **up_control** – Button control for the up (volume increase) button.
- **down_control** – Button control for the down (volume decrease) button.
- **select_control** – Button control for the select (play/pause) button.
- **select_hold_control** – Button control for holding the select button.
- **menu_control** – Button control for the menu button.
- **menu_hold_control** – Button control for holding the menu button.

Methods:

Attributes:

event_types

Attributes

`AppleRemote.event_types = ['on_button_press', 'on_button_release']`

Inherited members

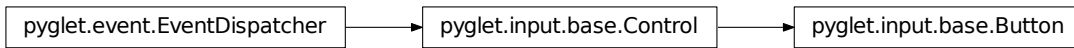
Methods

`AppleRemote.register_event_type` (*name*)

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters **name** (*str*) – Name of the event to register.



Button Class

class Button (*name*, *raw_name=None*)

A control whose value is boolean.

Methods:

Attributes:

<code>event_types</code>
<code>value</code>

Attributes

`Button.value`

Inherited members

Methods

`Button.register_event_type` (*name*)

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters *name* (*str*) – Name of the event to register.

Attributes

`Button.event_types` = ['on_change', 'on_press', 'on_release']



Control Class

class Control (*name*, *raw_name*=None)

Single value input provided by a device.

A control's value can be queried when the device is open. Event handlers can be attached to the control to be called when the value changes.

The *min* and *max* properties are provided as advertised by the device; in some cases the control's value will be outside this range.

Variables

- **name** – Name of the control, or None if unknown
- **raw_name** – Unmodified name of the control, as presented by the operating system; or None if unknown.
- **inverted** – If True, the value reported is actually inverted from what the device reported; usually this is to provide consistency across operating systems.

Methods:

Attributes:

event_types

value Current value of the control.

Attributes

`Control.event_types = ['on_change', 'on_press', 'on_release']`

`Control.value`

Current value of the control.

The range of the value is device-dependent; for absolute controls the range is given by *min* and *max* (however the value may exceed this range); for relative controls the range is undefined.

Type float

Inherited members

Methods

`Control.register_event_type` (*name*)

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters *name* (*str*) – Name of the event to register.

pyglet.input.base.Device

Device Class**class Device** (*display*, *name*)

Input device.

Variables

- **display** – Display this device is connected to.
- **name** – Name of the device, as described by the device firmware.
- **manufacturer** – Name of the device manufacturer, or `None` if the information is not available.

pyglet.event.EventDispatcher

→

pyglet.input.base.Joystick

Joystick Class**class Joystick** (*device*)

High-level interface for joystick-like devices. This includes analogue and digital joysticks, gamepads, game controllers, and possibly even steering wheels and other input devices. There is unfortunately no way to distinguish between these different device types.

To use a joystick, first call *open*, then in your game loop examine the values of *x*, *y*, and so on. These values are normalized to the range [-1.0, 1.0].

To receive events when the value of an axis changes, attach an `on_joyaxis_motion` event handler to the joystick. The *Joystick* instance, axis name, and current value are passed as parameters to this event.

To handle button events, you should attach `on_joybutton_press` and `on_joy_button_release` event handlers to the joystick. Both the *Joystick* instance and the index of the changed button are passed as parameters to these events.

Alternately, you may attach event handlers to each individual button in *button_controls* to receive `on_press` or `on_release` events.

To use the hat switch, attach an `on_joyhat_motion` event handler to the joystick. The handler will be called with both the *hat_x* and *hat_y* values whenever the value of the hat switch changes.

The device name can be queried to get the name of the joystick.

Variables

- **device** – The underlying device used by this joystick interface.
- **x** – Current X (horizontal) value ranging from -1.0 (left) to 1.0 (right).
- **y** – Current y (vertical) value ranging from -1.0 (top) to 1.0 (bottom).

- **z** – Current Z value ranging from -1.0 to 1.0. On joysticks the Z value is usually the throttle control. On game controllers the Z value is usually the secondary thumb vertical axis.
- **rx** – Current rotational X value ranging from -1.0 to 1.0.
- **ry** – Current rotational Y value ranging from -1.0 to 1.0.
- **rz** – Current rotational Z value ranging from -1.0 to 1.0. On joysticks the RZ value is usually the twist of the stick. On game controllers the RZ value is usually the secondary thumb horizontal axis.
- **hat_x** – Current hat (POV) horizontal position; one of -1 (left), 0 (centered) or 1 (right).
- **hat_y** – Current hat (POV) vertical position; one of -1 (bottom), 0 (centered) or 1 (top).
- **buttons** – List of boolean values representing current states of the buttons. These are in order, so that button 1 has value at `buttons[0]`, and so on.
- **x_control** – Underlying control for *x* value, or `None` if not available.
- **y_control** – Underlying control for *y* value, or `None` if not available.
- **z_control** – Underlying control for *z* value, or `None` if not available.
- **rx_control** – Underlying control for *rx* value, or `None` if not available.
- **ry_control** – Underlying control for *ry* value, or `None` if not available.
- **rz_control** – Underlying control for *rz* value, or `None` if not available.
- **hat_x_control** – Underlying control for *hat_x* value, or `None` if not available.
- **hat_y_control** – Underlying control for *hat_y* value, or `None` if not available.
- **button_controls** – Underlying controls for *buttons* values.

Methods:

Attributes:

[`event_types`](#)

Attributes

`Joystick.event_types = ['on_joyaxis_motion', 'on_joybutton_press', 'on_joybutton_release', 'on_joyhat_motion']`

Inherited members**Methods**

`Joystick.register_event_type(name)`

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters **name** (*str*) – Name of the event to register.



RelativeAxis Class

class RelativeAxis (*name*, *raw_name=None*)

An axis whose value represents a relative change from the previous value.

Methods:

Attributes:

<i>RX</i>	Name of the rotational-X axis control
<i>RY</i>	Name of the rotational-Y axis control
<i>RZ</i>	Name of the rotational-Z axis control
<i>WHEEL</i>	Name of the scroll wheel control
<i>X</i>	Name of the horizontal axis control
<i>Y</i>	Name of the vertical axis control
<i>Z</i>	Name of the Z axis control.
<i>event_types</i>	
<i>value</i>	

Attributes

`RelativeAxis.RX = 'rx'`

Name of the rotational-X axis control

`RelativeAxis.RY = 'ry'`

Name of the rotational-Y axis control

`RelativeAxis.RZ = 'rz'`

Name of the rotational-Z axis control

`RelativeAxis.WHEEL = 'wheel'`

Name of the scroll wheel control

`RelativeAxis.X = 'x'`

Name of the horizontal axis control

`RelativeAxis.Y = 'y'`

Name of the vertical axis control

`RelativeAxis.Z = 'z'`

Name of the Z axis control.

`RelativeAxis.value`

Inherited members

Methods

`RelativeAxis.register_event_type(name)`

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters `name` (*str*) – Name of the event to register.

Attributes

`RelativeAxis.event_types = ['on_change', 'on_press', 'on_release']`

pyglet.input.base.Tablet

Tablet Class

class `Tablet`

High-level interface to tablet devices.

Unlike other devices, tablets must be opened for a specific window, and cannot be opened exclusively. The *open* method returns a *TabletCanvas* object, which supports the events provided by the tablet.

Currently only one tablet device can be used, though it can be opened on multiple windows. If more than one tablet is connected, the behaviour is undefined.

pyglet.event.EventDispatcher

pyglet.input.base.TabletCanvas

TabletCanvas Class

class `TabletCanvas` (*window*)

Event dispatcher for tablets.

Use *Tablet.open* to obtain this object for a particular tablet device and window. Events may be generated even if the tablet stylus is outside of the window; this is operating-system dependent.

The events each provide the *TabletCursor* that was used to generate the event; for example, to distinguish between a stylus and an eraser. Only one cursor can be used at a time, otherwise the results are undefined.

Variables `window` – The window on which this tablet was opened.

Methods:

Attributes:

event_types

Attributes`TabletCanvas.event_types = ['on_enter', 'on_leave', 'on_motion']`**Inherited members****Methods**`TabletCanvas.register_event_type(name)`

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters *name* (*str*) – Name of the event to register.

pyglet.input.base.TabletCursor

TabletCursor* Class*class** `TabletCursor` (*name*)

A distinct cursor used on a tablet.

Most tablets support at least a *stylus* and an *erasor* cursor; this object is used to distinguish them when tablet events are generated.**Variables** *name* – Name of the cursor.

DeviceException

DeviceExclusiveException

DeviceOpenException

Exceptions

pyglet.input.base.DeviceException

DeviceException

Exception defined in `pyglet.input.base`

exception DeviceException

pyglet.input.base.DeviceException



pyglet.input.base.DeviceExclusiveException

DeviceExclusiveException Exception defined in `pyglet.input.base`

exception DeviceExclusiveException

pyglet.input.base.DeviceException



pyglet.input.base.DeviceOpenException

DeviceOpenException Exception defined in `pyglet.input.base`

exception DeviceOpenException

Variables

division = `_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)`

Notes	Defined
	<ul style="list-style-type: none"> <code>sys</code>

pyglet.input.evdev_constants Event constants from `/usr/include/linux/input.h`

Classes

Exceptions

Functions

<code>get_apple_remote([display])</code>	Get the Apple remote control device.
<code>get_devices([display])</code>	Get a list of all attached input devices.
<code>get_joysticks([display])</code>	Get a list of attached joysticks.
<code>get_tablets([display])</code>	Get a list of tablets.

`get_apple_remote` Function Defined in `pyglet.input`

`get_apple_remote` (*display=None*)

Get the Apple remote control device.

The Apple remote is the small white 6-button remote control that accompanies most recent Apple desktops and laptops. The remote can only be used with Mac OS X.

Parameters **display** (`Display`) – Currently ignored.

Return type *AppleRemote*

Returns The remote device, or *None* if the computer does not support it.

`get_devices` Function Defined in `pyglet.input`

`get_devices` (*display=None*)

Get a list of all attached input devices.

Parameters **display** (`Display`) – The display device to query for input devices. Ignored on Mac OS X and Windows. On Linux, defaults to the default display device.

Return type list of *Device*

`get_joysticks` Function Defined in `pyglet.input`

`get_joysticks` (*display=None*)

Get a list of attached joysticks.

Parameters **display** (`Display`) – The display device to query for input devices. Ignored on Mac OS X and Windows. On Linux, defaults to the default display device.

Return type list of *Joystick*

`get_tablets` Function Defined in `pyglet.input`

`get_tablets` (*display=None*)

Get a list of tablets.

This function may return a valid tablet device even if one is not attached (for example, it is not possible on Mac OS X to determine if a tablet device is connected). Despite returning a list of tablets, pyglet does not currently support multiple tablets, and the behaviour is undefined if more than one is attached.

Parameters **display** (`Display`) – The display device to query for input devices. Ignored on Mac OS X and Windows. On Linux, defaults to the default display device.

Return type list of *Tablet*

Variables

`absolute_import = _Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)`

Notes

Defined

- `sys`

pyglet.media

Audio and video playback.

pyglet can play WAV files, and if AVbin is installed, many other audio and video formats.

Playback is handled by the *Player* class, which reads raw data from *Source* objects and provides methods for pausing, seeking, adjusting the volume, and so on. The *Player* class implements the best available audio device (currently, only OpenAL is supported):

```
player = Player()
```

A *Source* is used to decode arbitrary audio and video files. It is associated with a single player by “queuing” it:

```
source = load('background_music.mp3')
player.queue(source)
```

Use the *Player* to control playback.

If the source contains video, the *Source.video_format* attribute will be non-None, and the *Player.texture* attribute will contain the current video image synchronised to the audio.

Decoding sounds can be processor-intensive and may introduce latency, particularly for short sounds that must be played quickly, such as bullets or explosions. You can force such sounds to be decoded and retained in memory rather than streamed from disk by wrapping the source in a *StaticSource*:

```
bullet_sound = StaticSource(load('bullet.wav'))
```

The other advantage of a *StaticSource* is that it can be queued on any number of players, and so played many times simultaneously.

pyglet relies on Python’s garbage collector to release resources when a player has finished playing a source. In this way some operations that could affect the application performance can be delayed.

The player provides a *Player.delete()* method that can be used to release resources immediately. Also an explicit call to `gc.collect()` can be used to collect unused resources.

Modules

drivers

Drivers for playing back media.

Continued on next page

Table 2.134 – continued from previous page

<i>events</i>	
<i>exceptions</i>	
<i>listener</i>	
<i>player</i>	
<i>sources</i>	Sources for media playback.
<i>threads</i>	

pyglet.media.drivers Drivers for playing back media.

<i>base</i>
<i>silent</i>

Modules

pyglet.media.drivers.base

<i>AbstractAudioDriver</i>	
<i>AbstractAudioPlayer</i>	Base class for driver audio players.

Classes

pyglet.media.drivers.base.AbstractAudioDriver

AbstractAudioDriver Class

class AbstractAudioDriver

pyglet.media.drivers.base.AbstractAudioPlayer

AbstractAudioPlayer Class

class AbstractAudioPlayer (*source_group*, *player*)
Base class for driver audio players.

pyglet.media.drivers.silent

<i>EventBuffer</i>	Buffer for events from audio data
<i>SilentAudioBuffer</i>	Buffer for silent audio packets
<i>SilentAudioDriver</i>	
<i>SilentAudioPacket</i>	
<i>SilentAudioPlayerPacketConsumer</i>	
<i>SilentTimeAudioPlayer</i>	

Classes

pyglet.media.drivers.silent.EventBuffer

EventBuffer Class

class EventBuffer

Buffer for events from audio data

pyglet.media.drivers.silent.SilentAudioBuffer

SilentAudioBuffer Class

class SilentAudioBuffer

Buffer for silent audio packets

pyglet.media.drivers.base.AbstractAudioDriver

pyglet.media.drivers.silent.SilentAudioDriver



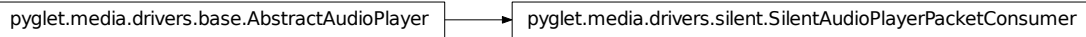
SilentAudioDriver Class

class SilentAudioDriver

pyglet.media.drivers.silent.SilentAudioPacket

SilentAudioPacket Class

class **SilentAudioPacket** (*timestamp, duration*)



SilentAudioPlayerPacketConsumer Class

class **SilentAudioPlayerPacketConsumer** (*source_group, player*)



SilentTimeAudioPlayer Class

class **SilentTimeAudioPlayer** (*source_group, player*)

create_audio_driver()

Functions

***create_audio_driver* Function** Defined in *pyglet.media.drivers.silent*

create_audio_driver ()

Variables

division = **_Feature**((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)

print_function = **_Feature**((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)

Defined

Notes

- pyglet
- time

[`get_audio_driver\(\)`](#)
[`get_silent_audio_driver\(\)`](#)

Functions

`get_audio_driver` Function Defined in `pyglet.media.drivers`

`get_audio_driver()`

`get_silent_audio_driver` Function Defined in `pyglet.media.drivers`

`get_silent_audio_driver()`

Variables

`absolute_import = _Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)`

`print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)`

Notes	Defined
	<ul style="list-style-type: none"> pyglet

`pyglet.media.events`

[`MediaEvent`](#)

Classes

`pyglet.media.events.MediaEvent`

MediaEvent Class

class `MediaEvent` (*timestamp, event, *args*)

Notes	Defined
	<ul style="list-style-type: none"> pyglet time

`pyglet.media.exceptions`

[*CannotSeekException*](#)
[*MediaException*](#)
[*MediaFormatException*](#)

Exceptions



CannotSeekException Exception defined in `pyglet.media.exceptions`
exception `CannotSeekException`

pyglet.media.exceptions.MediaException

MediaException

Exception defined in `pyglet.media.exceptions`
exception `MediaException`



MediaFormatException Exception defined in `pyglet.media.exceptions`
exception `MediaFormatException`

`pyglet.media.listener`

[*AbstractListener*](#) The listener properties for positional audio.

Classes

pyglet.media.listener.AbstractListener

AbstractListener Class

class **AbstractListener**

The listener properties for positional audio.

You can obtain the singleton instance of this class by calling *AbstractAudioDriver.get_listener*.

Attributes:

<i>forward_orientation</i>	A vector giving the direction the listener is facing.
<i>position</i>	The position of the listener in 3D space.
<i>up_orientation</i>	A vector giving the “up” orientation of the listener.
<i>volume</i>	The master volume for sound playback.

Attributes

`AbstractListener.forward_orientation`

A vector giving the direction the listener is facing.

The orientation is given as a tuple of floats (x, y, z), and has no unit. The forward orientation should be orthogonal to the up orientation.

Type 3-tuple of float

`AbstractListener.position`

The position of the listener in 3D space.

The position is given as a tuple of floats (x, y, z). The unit defaults to meters, but can be modified with the listener properties.

Type 3-tuple of float

`AbstractListener.up_orientation`

A vector giving the “up” orientation of the listener.

The orientation is given as a tuple of floats (x, y, z), and has no unit. The up orientation should be orthogonal to the forward orientation.

Type 3-tuple of float

`AbstractListener.volume`

The master volume for sound playback.

All sound volumes are multiplied by this master volume before being played. A value of 0 will silence playback (but still consume resources). The nominal volume is 1.0.

Type float

`pyglet.media.player`

<i>Player</i>	High-level sound and video player.
<i>PlayerGroup</i>	Group of players that can be played and paused simultaneously.

Classes



Player Class

class **Player**

High-level sound and video player.

Methods:

Attributes:

<i>cone_inner_angle</i>
<i>cone_orientation</i>
<i>cone_outer_angle</i>
<i>cone_outer_gain</i>
<i>event_types</i>
<i>max_distance</i>
<i>min_distance</i>
<i>pitch</i>
<i>playing</i>
<i>position</i>
<i>source</i>
<i>time</i>
<i>volume</i>

Attributes

`Player.cone_inner_angle`

`Player.cone_orientation`

`Player.cone_outer_angle`

`Player.cone_outer_gain`

`Player.event_types = ['on_eos', 'on_player_eos', 'on_source_group_eos']`

`Player.max_distance`

`Player.min_distance`

`Player.pitch`

`Player.playing`

`Player.position`

`Player.source`

`Player.time`

`Player.volume`

Inherited members

Methods

`Player.register_event_type(name)`

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters `name` (*str*) – Name of the event to register.

pyglet.media.player.PlayerGroup

PlayerGroup Class

class `PlayerGroup(players)`

Group of players that can be played and paused simultaneously.

Variables `players` – Players in this group.

Variables

`division = _Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)`

`print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)`

Notes	Defined
	<ul style="list-style-type: none">pyglet

`pyglet.media.sources` Sources for media playback.

Continued on next page

Table 2.147 – continued from previous page

<i>base</i>	
<i>loader</i>	
<i>procedural</i>	
<i>riff</i>	Simple Python-only RIFF reader, supports uncompressed WAV files.

Modules

`pyglet.media.sources.base`

<i>AudioData</i>	A single packet of audio data.
<i>AudioFormat</i>	Audio details.
<i>Source</i>	An audio and/or video source.
<i>SourceGroup</i>	Read data from a queue of sources, with support for looping.
<i>SourceInfo</i>	Source metadata information.
<i>StaticMemorySource</i>	Helper class for default implementation of <i>StaticSource</i> .
<i>StaticSource</i>	A source that has been completely decoded in memory.
<i>StreamingSource</i>	A source that is decoded as it is being played, and can only be queued once.
<i>VideoFormat</i>	Video details.

Classes

`pyglet.media.sources.base.AudioData`

AudioData Class

class **AudioData** (*data, length, timestamp, duration, events*)

A single packet of audio data.

This class is used internally by pyglet.

Variables

- ***data*** – Sample data.
- ***length*** – Size of sample data, in bytes.
- ***timestamp*** – Time of the first sample, in seconds.
- ***duration*** – Total data duration, in seconds.
- ***events*** – List of events contained within this packet. Events are timestamped relative to this audio packet.

pyglet.media.sources.base.AudioFormat

AudioFormat Class

class AudioFormat (*channels, sample_size, sample_rate*)

Audio details.

An instance of this class is provided by sources with audio tracks. You should not modify the fields, as they are used internally to describe the format of data provided by the source.

Variables

- **channels** – The number of channels: 1 for mono or 2 for stereo (pyglet does not yet support surround-sound sources).
- **sample_size** – Bits per sample; only 8 or 16 are supported.
- **sample_rate** – Samples per second (in Hertz).

pyglet.media.sources.base.Source

Source Class

class Source

An audio and/or video source.

Variables

- **audio_format** – Format of the audio in this source, or None if the source is silent.
- **video_format** – Format of the video in this source, or None if there is no video.
- **info** – Source metadata such as title, artist, etc; or None if the information is not available.

Since: pyglet 1.2

Attributes:

<i>audio_format</i>	
<i>duration</i>	The length of the source, in seconds.
<i>info</i>	
<i>video_format</i>	

Attributes

`Source.audio_format = None`

`Source.duration`

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is `None`.

Read-only.

Type float

`Source.info = None`

`Source.video_format = None`

`pyglet.media.sources.base.SourceGroup`

SourceGroup Class

class `SourceGroup` (*audio_format*, *video_format*)

Read data from a queue of sources, with support for looping. All sources must share the same audio format.

Variables *audio_format* – Required audio format for queued sources.

Attributes:

loop Loop the current source indefinitely or until *next* is called.

Attributes

`SourceGroup.loop`

Loop the current source indefinitely or until *next* is called. Initially `False`.

Type bool

`pyglet.media.sources.base.SourceInfo`

SourceInfo Class

class `SourceInfo`

Source metadata information.

Fields are the empty string or zero if the information is not available.

Variables

- **title** – Title
- **author** – Author

- *copyright* – Copyright statement
- *comment* – Comment
- *album* – Album name
- *year* – Year
- *track* – Track number
- *genre* – Genre

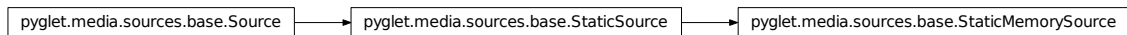
Note: Since pyglet 1.2

Attributes:

<i>album</i>
<i>author</i>
<i>comment</i>
<i>copyright</i>
<i>genre</i>
<i>title</i>
<i>track</i>
<i>year</i>

Attributes

```
SourceInfo.album = ''
SourceInfo.author = ''
SourceInfo.comment = ''
SourceInfo.copyright = ''
SourceInfo.genre = ''
SourceInfo.title = ''
SourceInfo.track = 0
SourceInfo.year = 0
```



***StaticMemorySource* Class**

class StaticMemorySource (*data*, *audio_format*)

Helper class for default implementation of *StaticSource*. Do not use directly.

Attributes:

<i>audio_format</i>	
<i>duration</i>	The length of the source, in seconds.

Continued on next page

Table 2.152 – continued from previous page

info
video_format

Inherited members**Attributes**

`StaticMemorySource.audio_format = None`

`StaticMemorySource.duration`

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is `None`.

Read-only.

Type float

`StaticMemorySource.info = None`

`StaticMemorySource.video_format = None`

***StaticSource* Class**

class `StaticSource` (*source*)

A source that has been completely decoded in memory. This source can be queued onto multiple players any number of times.

Attributes:

audio_format	
duration	The length of the source, in seconds.
info	
video_format	

Inherited members**Attributes**

`StaticSource.audio_format = None`

`StaticSource.duration`

The length of the source, in seconds.

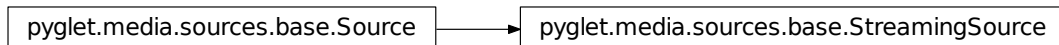
Not all source durations can be determined; in this case the value is `None`.

Read-only.

Type float

`StaticSource.info = None`

`StaticSource.video_format = None`



StreamingSource Class

class StreamingSource

A source that is decoded as it is being played, and can only be queued once.

Attributes:

<hr/>	
<code>audio_format</code>	
<code>duration</code>	The length of the source, in seconds.
<code>info</code>	
<code>is_queued</code>	Determine if this source has been queued on a <i>Player</i> yet.
<code>video_format</code>	
<hr/>	

Attributes

`StreamingSource.is_queued`

Determine if this source has been queued on a *Player* yet.

Read-only.

Type bool

Inherited members

Attributes

`StreamingSource.audio_format = None`

`StreamingSource.duration`

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is `None`.

Read-only.

Type float

```
StreamingSource.info = None
StreamingSource.video_format = None
```

pyglet.media.sources.base.VideoFormat

VideoFormat Class

class VideoFormat (*width, height, sample_aspect=1.0*)
 Video details.

An instance of this class is provided by sources with a video track. You should not modify the fields.

Note that the sample aspect has no relation to the aspect ratio of the video image. For example, a video image of 640x480 with sample aspect 2.0 should be displayed at 1280x480. It is the responsibility of the application to perform this scaling.

Variables

- **width** – Width of video image, in pixels.
- **height** – Height of video image, in pixels.
- **sample_aspect** – Aspect ratio (width over height) of a single video pixel.
- **frame_rate** – Frame rate (frames per second) of the video. AVbin 8 or later is required, otherwise the frame rate will be `None`. **Since:** pyglet 1.2.

Variables

```
division = _Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)
```

Defined

- Notes**
- `pyglet`
 - `sys`

`pyglet.media.sources.loader`

AVbinSourceLoader

AbstractSourceLoader

RIFFSourceLoader

Classes



AVbinSourceLoader Class
class AVbinSourceLoader

pyglet.media.sources.loader.AbstractSourceLoader

AbstractSourceLoader Class

class AbstractSourceLoader



RIFFSourceLoader Class
class RIFFSourceLoader

<u><code>get_source_loader()</code></u>	
<u><code>have_avbin()</code></u>	
<u><code>load(filename[, file, streaming])</code></u>	Load a source from a file.

Functions

get_source_loader Function Defined in `pyglet.media.sources.loader`

get_source_loader()

have_avbin Function Defined in `pyglet.media.sources.loader`

have_avbin()

load Function Defined in `pyglet.media.sources.loader`

load (*filename*, *file=None*, *streaming=True*)

Load a source from a file.

Currently the *file* argument is not supported; media files must exist as real paths.

Parameters

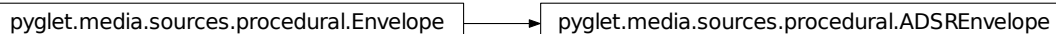
- **filename** (*str*) – Filename of the media file to load.
- **file** (*file-like object*) – Not yet supported.
- **streaming** (*bool*) – If False, a *StaticSource* will be returned; otherwise (default) a *StreamingSource* is created.

Return type *Source***Variables****print_function** = `_Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)`**Defined**
Notes

- pyglet

pyglet.media.sources.procedural

<i>ADSREnvelope</i>	A four part Attack, Decay, Suspend, Release envelope.
<i>Digital</i>	A procedurally generated guitar-like waveform.
<i>Envelope</i>	Base class for ProceduralSource amplitude envelopes.
<i>FM</i>	A procedurally generated FM waveform.
<i>FlatEnvelope</i>	A flat envelope, providing basic amplitude setting.
<i>LinearDecayEnvelope</i>	A linearly decaying envelope.
<i>ProceduralSource</i>	Base class for procedurally defined and generated waveforms.
<i>Sawtooth</i>	A procedurally generated sawtooth waveform.
<i>Silence</i>	A silent waveform.
<i>Sine</i>	A procedurally generated sinusoid waveform.
<i>Square</i>	A procedurally generated square (or pulse) waveform.
<i>Triangle</i>	A procedurally generated triangle waveform.
<i>WhiteNoise</i>	A white noise, random waveform.

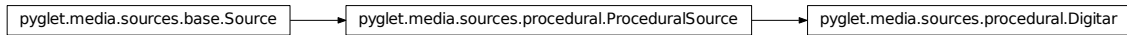
Classes**ADSREnvelope Class****class ADSREnvelope** (*attack, decay, release, sustain_amplitude=0.5*)

A four part Attack, Decay, Suspend, Release envelope.

This is a four part ADSR envelope. The attack, decay, and release parameters should be provided in seconds. For example, a value of 0.1 would be 100ms. The sustain_amplitude parameter affects the sustain volume. This defaults to a value of 0.5, but can be provided on a scale from 0.0 to 1.0.

Parameters

- **attack** (*float*) – The attack time, in seconds.
- **decay** (*float*) – The decay time, in seconds.
- **release** (*float*) – The release time, in seconds.
- **sustain_amplitude** (*float*) – The sustain amplitude (volume), from 0.0 to 1.0.



Digital Class

class Digital (*duration, frequency=440, decay=0.996, **kwargs*)

A procedurally generated guitar-like waveform.

A guitar-like waveform, based on the Karplus-Strong algorithm. The sound is similar to a plucked guitar string. The resulting sound decays over time, and so the actual length will vary depending on the frequency. Lower frequencies require a longer *length* parameter to prevent cutting off abruptly.

Parameters

- **duration** (*float*) – The length, in seconds, of audio that you wish to generate.
- **frequency** (*int*) – The frequency, in Hz of the waveform you wish to produce.
- **decay** (*float*) – The decay rate of the effect. Defaults to 0.996.
- **sample_rate** (*int*) – Audio samples per second. (CD quality is 44100).
- **sample_size** (*int*) – The bit precision. Must be either 8 or 16.

Attributes:

audio_format	
duration	The length of the source, in seconds.
envelope	
info	
video_format	

Inherited members

Attributes

Digital.**audio_format** = None

Digital.**duration**

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is None.

Read-only.

Type float


```

Digitalar.envelope
Digitalar.info = None
Digitalar.video_format = None

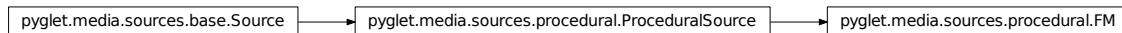
```

```
pyglet.media.sources.procedural.Envelope
```

Envelope Class

class **Envelope**

Base class for ProceduralSource amplitude envelopes.



FM Class

class FM(*duration*, *carrier*=440, *modulator*=440, *mod_index*=1, ***kwargs*)

A procedurally generated FM waveform.

This is a simplistic frequency modulated waveform, based on the concepts by John Chowning. Basic sine waves are used for both frequency carrier and modulator inputs, of which the frequencies can be provided. The modulation index, or amplitude, can also be adjusted.

Parameters

- **duration** (*float*) – The length, in seconds, of audio that you wish to generate.
- **carrier** (*int*) – The carrier frequency, in Hz.
- **modulator** (*int*) – The modulator frequency, in Hz.
- **mod_index** (*int*) – The modulation index.
- **sample_rate** (*int*) – Audio samples per second. (CD quality is 44100).
- **sample_size** (*int*) – The bit precision. Must be either 8 or 16.

Attributes:

audio_format	
duration	The length of the source, in seconds.
envelope	
info	
video_format	

Inherited members

Attributes

`FM.audio_format = None`

`FM.duration`

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is `None`.

Read-only.

Type float

`FM.envelope`

`FM.info = None`

`FM.video_format = None`

`pyglet.media.sources.procedural.Envelope`

`pyglet.media.sources.procedural.FlatEnvelope`

FlatEnvelope Class

class `FlatEnvelope` (*amplitude=0.5*)

A flat envelope, providing basic amplitude setting.

Parameters **amplitude** (*float*) – The amplitude (volume) of the wave, from 0.0 to 1.0. Values outside of this range will be clamped.

`pyglet.media.sources.procedural.Envelope`

`pyglet.media.sources.procedural.LinearDecayEnvelope`

LinearDecayEnvelope Class

class `LinearDecayEnvelope` (*peak=1.0*)

A linearly decaying envelope.

This envelope linearly decays the amplitude from the peak value to 0, over the length of the waveform.

Parameters **peak** (*float*) – The Initial peak value of the envelope, from 0.0 to 1.0. Values outside of this range will be clamped.

`pyglet.media.sources.base.Source`

`pyglet.media.sources.procedural.ProceduralSource`

ProceduralSource Class

class ProceduralSource (*duration*, *sample_rate*=44800, *sample_size*=16, *envelope*=None)

Base class for procedurally defined and generated waveforms.

Parameters

- **duration** (*float*) – The length, in seconds, of audio that you wish to generate.
- **sample_rate** (*int*) – Audio samples per second. (CD quality is 44100).
- **sample_size** (*int*) – The bit precision. Must be either 8 or 16.

Attributes:

<code>audio_format</code>	
<code>duration</code>	The length of the source, in seconds.
<code>envelope</code>	
<code>info</code>	
<code>video_format</code>	

Attributes

`ProceduralSource.envelope`

Inherited members

Attributes

`ProceduralSource.audio_format = None`

`ProceduralSource.duration`

The length of the source, in seconds.

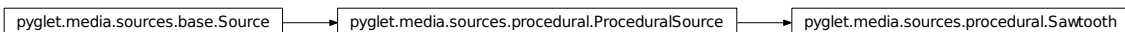
Not all source durations can be determined; in this case the value is None.

Read-only.

Type float

`ProceduralSource.info = None`

`ProceduralSource.video_format = None`



Sawtooth Class

class Sawtooth (*duration*, *frequency*=440, ***kwargs*)

A procedurally generated sawtooth waveform.

Parameters

- **duration** (*float*) – The length, in seconds, of audio that you wish to generate.
- **frequency** (*int*) – The frequency, in Hz of the waveform you wish to produce.

- **sample_rate** (*int*) – Audio samples per second. (CD quality is 44100).
- **sample_size** (*int*) – The bit precision. Must be either 8 or 16.

Attributes:

audio_format	
duration	The length of the source, in seconds.
envelope	
info	
video_format	

Inherited members**Attributes**

`Sawtooth.audio_format = None`

`Sawtooth.duration`

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is None.

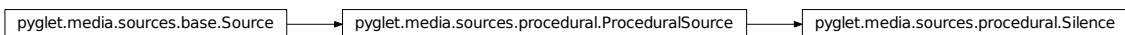
Read-only.

Type float

`Sawtooth.envelope`

`Sawtooth.info = None`

`Sawtooth.video_format = None`

**Silence Class**

class Silence (*duration, sample_rate=44800, sample_size=16, envelope=None*)

A silent waveform.

Attributes:

audio_format	
duration	The length of the source, in seconds.
envelope	
info	
video_format	

Inherited members

Attributes

`Silence.audio_format = None`

`Silence.duration`

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is `None`.

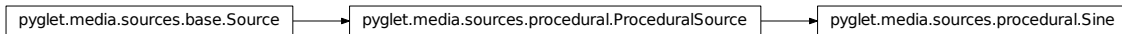
Read-only.

Type float

`Silence.envelope`

`Silence.info = None`

`Silence.video_format = None`



Sine Class

class `Sine` (*duration*, *frequency*=440, ***kwargs*)

A procedurally generated sinusoid waveform.

Parameters

- **duration** (*float*) – The length, in seconds, of audio that you wish to generate.
- **frequency** (*int*) – The frequency, in Hz of the waveform you wish to produce.
- **sample_rate** (*int*) – Audio samples per second. (CD quality is 44100).
- **sample_size** (*int*) – The bit precision. Must be either 8 or 16.

Attributes:

<code>audio_format</code>	
<code>duration</code>	The length of the source, in seconds.
<code>envelope</code>	
<code>info</code>	
<code>video_format</code>	

Inherited members

Attributes

`Sine.audio_format = None`

`Sine.duration`

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is `None`.

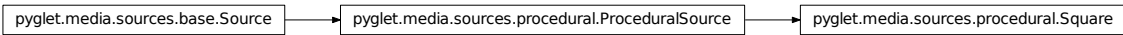
Read-only.

Type float

`Sine.envelope`

`Sine.info = None`

`Sine.video_format = None`



Square Class

class Square (*duration*, *frequency*=440, ***kwargs*)

A procedurally generated square (or pulse) waveform.

Parameters

- **duration** (*float*) – The length, in seconds, of audio that you wish to generate.
- **frequency** (*int*) – The frequency, in Hz of the waveform you wish to produce.
- **sample_rate** (*int*) – Audio samples per second. (CD quality is 44100).
- **sample_size** (*int*) – The bit precision. Must be either 8 or 16.

Attributes:

<code>audio_format</code>	
<code>duration</code>	The length of the source, in seconds.
<code>envelope</code>	
<code>info</code>	
<code>video_format</code>	

Inherited members

Attributes

`Square.audio_format = None`

`Square.duration`

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is None.

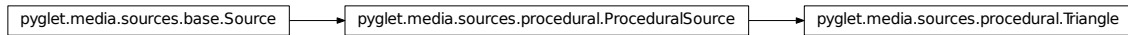
Read-only.

Type float

`Square.envelope`

`Square.info = None`

`Square.video_format = None`



Triangle Class

class Triangle (*duration*, *frequency=440*, ***kwargs*)

A procedurally generated triangle waveform.

Parameters

- **duration** (*float*) – The length, in seconds, of audio that you wish to generate.
- **frequency** (*int*) – The frequency, in Hz of the waveform you wish to produce.
- **sample_rate** (*int*) – Audio samples per second. (CD quality is 44100).
- **sample_size** (*int*) – The bit precision. Must be either 8 or 16.

Attributes:

audio_format	
duration	The length of the source, in seconds.
envelope	
info	
video_format	

Inherited members

Attributes

`Triangle.audio_format = None`

`Triangle.duration`

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is None.

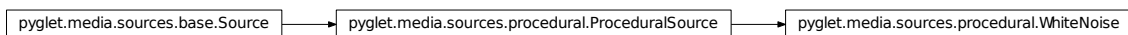
Read-only.

Type float

`Triangle.envelope`

`Triangle.info = None`

`Triangle.video_format = None`



WhiteNoise Class

class WhiteNoise (*duration*, *sample_rate*=44800, *sample_size*=16, *envelope*=None)
 A white noise, random waveform.

Attributes:

audio_format	
duration	The length of the source, in seconds.
envelope	
info	
video_format	

Inherited members

Attributes

`WhiteNoise.audio_format = None`

`WhiteNoise.duration`

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is None.

Read-only.

Type float

`WhiteNoise.envelope`

`WhiteNoise.info = None`

`WhiteNoise.video_format = None`

Variables

`division = _Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)`

Defined	
Notes	• math
	• os
	• random
	• struct

pyglet.media.sources.riff Simple Python-only RIFF reader, supports uncompressed WAV files.

<i>RIFFChunk</i>
<i>RIFFFile</i>
<i>RIFFForm</i>
<i>RIFFType</i>
Continued on next page

Table 2.167 – continued from previous page

<i>WaveDataChunk</i>
<i>WaveForm</i>
<i>WaveFormatChunk</i>
<i>WaveSource</i>

Classes

pyglet.media.sources.riff.RIFFChunk

RIFFChunk Class

class **RIFFChunk** (*file*, *name*, *length*, *offset*)

Attributes:

<i>header_fmt</i>
<i>header_length</i>

Attributes

`RIFFChunk.header_fmt = '<4sL'`

`RIFFChunk.header_length = 8`

pyglet.media.sources.riff.RIFFForm

pyglet.media.sources.riff.RIFFFile

RIFFFile Class

class **RIFFFile** (*file*)

pyglet.media.sources.riff.RIFFForm

RIFFForm Class

class **RIFFForm** (*file*, *offset*)



RIFFType Class

class **RIFFType** (**args*, ***kwargs*)

Attributes:

header_fmt
header_length

Inherited members

Attributes

`RIFFType.header_fmt = '<4sL'`

`RIFFType.header_length = 8`



WaveDataChunk Class

class **WaveDataChunk** (*file*, *name*, *length*, *offset*)

Attributes:

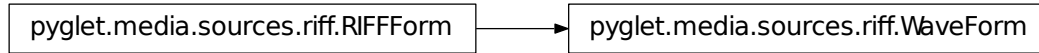
header_fmt
header_length

Inherited members

Attributes

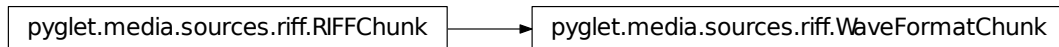
`WaveDataChunk.header_fmt = '<4sL'`

`WaveDataChunk.header_length = 8`



WaveForm Class

class WaveForm (*file, offset*)



WaveFormatChunk Class

class WaveFormatChunk (**args, **kwargs*)

Attributes:

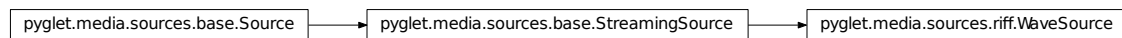
header_fmt
header_length

Inherited members

Attributes

`WaveFormatChunk.header_fmt = '<4sL'`

`WaveFormatChunk.header_length = 8`



WaveSource Class

class WaveSource (*filename, file=None*)

Attributes:

audio_format	
duration	The length of the source, in seconds.
Continued on next page	

Table 2.172 – continued from previous page

<code>info</code>	
<code>is_queued</code>	Determine if this source has been queued on a <i>Player</i> yet.
<code>video_format</code>	

Inherited members

Attributes

`WaveSource.audio_format = None`

`WaveSource.duration`

The length of the source, in seconds.

Not all source durations can be determined; in this case the value is `None`.

Read-only.

Type float

`WaveSource.info = None`

`WaveSource.is_queued`

Determine if this source has been queued on a *Player* yet.

Read-only.

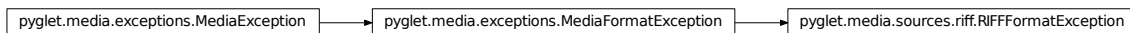
Type bool

`WaveSource.video_format = None`

`RIFFFormatException`

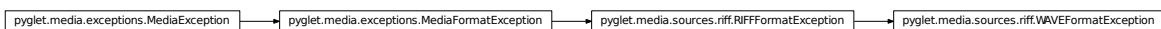
`WAVEFormatException`

Exceptions



RIFFFormatException Exception defined in `pyglet.media.sources.riff`

exception **`RIFFFormatException`**



WAVEFormatException Exception defined in `pyglet.media.sources.riff`

exception **`WAVEFormatException`**

Variables

IBM_FORMAT_ADPCM = 259

`int(x=0) -> integer int(x, base=10) -> integer`

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return `x.__int__()`. For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. `>>> int('0b100', base=0) 4`

IBM_FORMAT_ALAW = 258

`int(x=0) -> integer int(x, base=10) -> integer`

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return `x.__int__()`. For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. `>>> int('0b100', base=0) 4`

IBM_FORMAT_MULAW = 257

`int(x=0) -> integer int(x, base=10) -> integer`

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return `x.__int__()`. For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. `>>> int('0b100', base=0) 4`

WAVE_FORMAT_PCM = 1

`int(x=0) -> integer int(x, base=10) -> integer`

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return `x.__int__()`. For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. `>>> int('0b100', base=0) 4`

`division = _Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)`

Notes	Defined
	<ul style="list-style-type: none"> <code>struct</code>

Classes

`pyglet.media.threads`

<i>MediaThread</i>	A thread that cleanly exits on interpreter shutdown, and provides a sleep method that can be interrupted and a termination method.
<i>PlayerWorker</i>	Worker thread for refilling players.
<i>WorkerThread</i>	

Classes

pyglet.media.threads.MediaThread

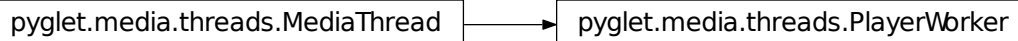
MediaThread Class

class MediaThread (*target=None*)

A thread that cleanly exits on interpreter shutdown, and provides a sleep method that can be interrupted and a termination method.

Variables

- **condition** – Lock condition on all instance variables.
- **stopped** – True if *stop* has been called.



PlayerWorker Class

class PlayerWorker

Worker thread for refilling players.



WorkerThread Class

class WorkerThread (*target=None*)

Variables

print_function = `_Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)`

Defined**Notes**

- atexit
- pyglet
- standard_library
- threading
- time

Notes**Defined**

- base
- loader
- procedural

pyglet.resource

Load application resources from a known path.

Loading resources by specifying relative paths to filenames is often problematic in Python, as the working directory is not necessarily the same directory as the application's script files.

This module allows applications to specify a search path for resources. Relative paths are taken to be relative to the application's `__main__` module. ZIP files can appear on the path; they will be searched inside. The resource module also behaves as expected when applications are bundled using py2exe or py2app.

As well as providing file references (with the *file* function), the resource module also contains convenience functions for loading images, textures, fonts, media and documents.

3rd party modules or packages not bound to a specific application should construct their own *Loader* instance and override the path to use the resources in the module's directory.

Path format

The resource path *path* (see also *Loader.__init__* and *Loader.path*) is a list of locations to search for resources. Locations are searched in the order given in the path. If a location is not valid (for example, if the directory does not exist), it is skipped.

Locations in the path beginning with an ampersand (“@” symbol) specify Python packages. Other locations specify a ZIP archive or directory on the filesystem. Locations that are not absolute are assumed to be relative to the script home. Some examples:

```
# Search just the `res` directory, assumed to be located alongside the
# main script file.
path = ['res']

# Search the directory containing the module `levels.level1`, followed
# by the `res/images` directory.
path = ['@levels.level1', 'res/images']
```

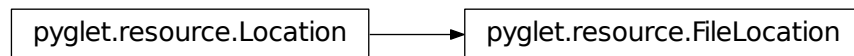
Paths are always case-sensitive and forward slashes are always used as path separators, even in cases when the filesystem or platform does not do this. This avoids a common programmer error when porting applications between platforms.

The default path is [' . ']. If you modify the path, you must call *reindex*.

Note: Since pyglet 1.1

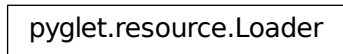
Classes

<i>FileLocation</i>	Location on the filesystem.
<i>Loader</i>	Load program resource files from disk.
<i>Location</i>	Abstract resource location.
<i>URLLocation</i>	Location on the network.
<i>ZIPLocation</i>	Location within a ZIP file.



FileLocation Class

class FileLocation (*path*)
Location on the filesystem.



Loader Class

class Loader (*path=None, script_home=None*)
Load program resource files from disk.

The loader contains a search path which can include filesystem directories, ZIP archives and Python packages.

Variables

- **path** – List of search locations. After modifying the path you must call the *reindex* method.
- **script_home** – Base resource location, defaulting to the location of the application script.

pyglet.resource.Location

Location Class

class Location

Abstract resource location.

Given a location, a file can be loaded from that location with the *open* method. This provides a convenient way to specify a path to load files from, and not necessarily have that path reside on the filesystem.



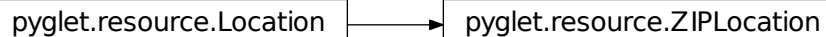
```
graph LR; A[pyglet.resource.Location] --> B[pyglet.resource.URLLocation]
```

URLLocation Class

class URLLocation (*base_url*)

Location on the network.

This class uses the `urlparse` and `urllib2` modules to open files on the network given a URL.



```
graph LR; A[pyglet.resource.Location] --> B[pyglet.resource.ZIPLocation]
```

ZIPLocation Class

class ZIPLocation (*zip, dir*)

Location within a ZIP file.

Exceptions

<code>ResourceNotFoundException</code> (name)	The named resource was not found on the search path.
---	--

pyglet.resource.ResourceNotFoundException

ResourceNotFoundException

Exception defined in `pyglet.resource`

exception ResourceNotFoundException (*name*)

The named resource was not found on the search path.

Functions

<code>get_script_home()</code>	Get the directory containing the program entry module.
<code>get_settings_path(name)</code>	Get a directory to save user preferences.

***get_script_home* Function** Defined in `pyglet.resource`

get_script_home ()

Get the directory containing the program entry module.

For ordinary Python scripts, this is the directory containing the `__main__` module. For executables created with py2exe the result is the directory containing the running executable file. For OS X bundles created using Py2App the result is the Resources directory within the running bundle.

If none of the above cases apply and the file for `__main__` cannot be determined the working directory is returned.

When the script is being run by a Python profiler, this function may return the directory where the profiler is running instead of the directory of the real script. To workaroud this behaviour the full path to the real script can be specified in `pyglet.resource.path`.

Return type str

***get_settings_path* Function** Defined in `pyglet.resource`

get_settings_path (*name*)

Get a directory to save user preferences.

Different platforms have different conventions for where to save user preferences, saved games, and settings. This function implements those conventions. Note that the returned path may not exist: applications should use `os.makedirs` to construct it if desired.

On Linux, a directory *name* in the user's configuration directory is returned (usually under `~/.config`).

On Windows (including under Cygwin) the *name* directory in the user's Application Settings directory is returned.

On Mac OS X the *name* directory under `~/Library/Application Support` is returned.

Parameters *name* (*str*) – The name of the application.

Return type str

Variables

add_font = <bound method Loader.add_font of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>
Add a font resource to the application.

Fonts not installed on the system must be added to pyglet before they can be used with *font.load*. Although the font is added with its filename using this function, it is loaded by specifying its family name. For example:

```
resource.add_font('action_man.ttf')
action_man = font.load('Action Man')
```

Parameters **name** (*str*) – Filename of the font resource to add.

animation = <bound method Loader.animation of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>
Load an animation with optional transformation.

Animations loaded from the same source but with different transformations will use the same textures.

Parameters

- **name** (*str*) – Filename of the animation source to load.
- **flip_x** (*bool*) – If True, the returned image will be flipped horizontally.
- **flip_y** (*bool*) – If True, the returned image will be flipped vertically.
- **rotate** (*int*) – The returned image will be rotated clockwise by the given number of degrees (a multiple of 90).

Return type *Animation*

attributed = <bound method Loader.attributed of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>
Load an attributed text document.

See *pyglet.text.formats.attributed* for details on this format.

Parameters **name** (*str*) – Filename of the attribute text resource to load.

Return type *FormattedDocument*

file = <bound method Loader.file of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>
Load a resource.

Parameters

- **name** (*str*) – Filename of the resource to load.
- **mode** (*str*) – Combination of r, w, a, b and t characters with the meaning as for the builtin *open* function.

Return type file object

get_cached_animation_names = <bound method Loader.get_cached_animation_names of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>
Get a list of animation filenames that have been cached.

This is useful for debugging and profiling only.

Return type list

Returns List of str

get_cached_image_names = <bound method Loader.get_cached_image_names of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>
Get a list of image filenames that have been cached.

This is useful for debugging and profiling only.

Return type list

Returns List of str

get_cached_texture_names = <bound method Loader.get_cached_texture_names of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>

Get the names of textures currently cached.

Return type list of str

get_texture_bins = <bound method Loader.get_texture_bins of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>

Get a list of texture bins in use.

This is useful for debugging and profiling only.

Return type list

Returns List of *TextureBin*

html = <bound method Loader.html of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>

Load an HTML document.

Parameters **name** (*str*) – Filename of the HTML resource to load.

Return type *FormattedDocument*

image = <bound method Loader.image of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>

Load an image with optional transformation.

This is similar to *texture*, except the resulting image will be packed into a *TextureBin* if it is an appropriate size for packing. This is more efficient than loading images into separate textures.

Parameters

- **name** (*str*) – Filename of the image source to load.
- **flip_x** (*bool*) – If True, the returned image will be flipped horizontally.
- **flip_y** (*bool*) – If True, the returned image will be flipped vertically.
- **rotate** (*int*) – The returned image will be rotated clockwise by the given number of degrees (a multiple of 90).
- **atlas** (*bool*) – If True, the image will be loaded into an atlas managed by pyglet. If atlas loading is not appropriate for specific texturing reasons (e.g. border control is required) then set this argument to False.

Return type *Texture*

Returns A complete texture if the image is large or not in an atlas, otherwise a *TextureRegion* of a texture atlas.

location = <bound method Loader.location of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>

Get the location of a resource.

This method is useful for opening files referenced from a resource. For example, an HTML file loaded as a resource might reference some images. These images should be located relative to the HTML file, not looked up individually in the loader's path.

Parameters **name** (*str*) – Filename of the resource to locate.

Return type *Location*

media = <bound method Loader.media of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>

Load a sound or video resource.

The meaning of *streaming* is as for *media.load*. Compressed sources cannot be streamed (that is, video and compressed audio cannot be streamed from a ZIP archive).

Parameters

- **name** (*str*) – Filename of the media source to load.
- **streaming** (*bool*) – True if the source should be streamed from disk, False if it should be entirely decoded into memory immediately.

Return type *media.Source*

path = ['.']

Default resource search path.

Locations in the search path are searched in order and are always case-sensitive. After changing the path you must call *reindex*.

See the module documentation for details on the path format.

Type list of str

reindex = <bound method Loader.reindex of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>

Refresh the file index.

You must call this method if *path* is changed or the filesystem layout changes.

text = <bound method Loader.text of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>

Load a plain text document.

Parameters **name** (*str*) – Filename of the plain text resource to load.

Return type *UnformattedDocument*

texture = <bound method Loader.texture of <pyglet.resource._DefaultLoader object at 0x7fa55279d940>>

Load a texture.

The named image will be loaded as a single OpenGL texture. If the dimensions of the image are not powers of 2 a *TextureRegion* will be returned.

Parameters **name** (*str*) – Filename of the image resource to load.

Return type *Texture*

Notes**Defined**

- os
- pyglet
- standard_library
- sys
- weakref
- zipfile

pyglet.sprite

Display positioned, scaled and rotated images.

A sprite is an instance of an image displayed on-screen. Multiple sprites can display the same image at different positions on the screen. Sprites can also be scaled larger or smaller, rotated at any angle and drawn at a fractional opacity.

The following complete example loads a "ball.png" image and creates a sprite for that image. The sprite is then drawn in the window's draw event handler:

```
import pyglet

ball_image = pyglet.image.load('ball.png')
ball = pyglet.sprite.Sprite(ball_image, x=50, y=50)

window = pyglet.window.Window()

@window.event
def on_draw():
    ball.draw()

pyglet.app.run()
```

The sprite can be moved by modifying the `x` and `y` properties. Other properties determine the sprite's rotation, scale and opacity.

By default sprite coordinates are restricted to integer values to avoid sub-pixel artifacts. If you require to use floats, for example for smoother animations, you can set the `subpixel` parameter to `True` when creating the sprite (:since: pyglet 1.2).

The sprite's positioning, rotation and scaling all honor the original image's anchor (`anchor_x`, `anchor_y`).

Drawing multiple sprites

Sprites can be "batched" together and drawn at once more quickly than if each of their `draw` methods were called individually. The following example creates one hundred ball sprites and adds each of them to a *Batch*. The entire batch of sprites is then drawn in one call:

```
batch = pyglet.graphics.Batch()

ball_sprites = []
for i in range(100):
    x, y = i * 10, 50
    ball_sprites.append(pyglet.sprite.Sprite(ball_image, x, y, batch=batch))

@window.event
def on_draw():
    batch.draw()
```

Sprites can be freely modified in any way even after being added to a batch, however a sprite can belong to at most one batch. See the documentation for *pyglet.graphics* for more details on batched rendering, and grouping of sprites within batches.

Note: Since pyglet 1.1

Classes

<i>Sprite</i>	Instance of an on-screen image.
<i>SpriteGroup</i>	Shared sprite rendering group.



Sprite Class

class Sprite (*img, x=0, y=0, blend_src=770, blend_dest=771, batch=None, group=None, usage='dynamic', subpixel=False*)

Instance of an on-screen image.

See the module documentation for usage.

Methods:

Attributes:

<i>batch</i>	Graphics batch.
<i>color</i>	Blend color.
<i>event_types</i>	
<i>group</i>	Parent graphics group.
<i>height</i>	Scaled height of the sprite.
<i>image</i>	Image or animation to display.
<i>opacity</i>	Blend opacity.
<i>position</i>	The (x, y) coordinates of the sprite, as a tuple.
<i>rotation</i>	Clockwise rotation of the sprite, in degrees.
<i>scale</i>	Scaling factor.
<i>scale_x</i>	Horizontal scaling factor.
<i>scale_y</i>	Vertical scaling factor.
<i>visible</i>	True if the sprite will be drawn.
<i>width</i>	Scaled width of the sprite.
<i>x</i>	X coordinate of the sprite.
<i>y</i>	Y coordinate of the sprite.

Attributes

Sprite.batch

Graphics batch.

The sprite can be migrated from one batch to another, or removed from its batch (for individual drawing). Note that this can be an expensive operation.

Type Batch

Sprite.color

Blend color.

This property sets the color of the sprite's vertices. This allows the sprite to be drawn with a color tint.

The color is specified as an RGB tuple of integers '(red, green, blue)'. Each color component must be in the range 0 (dark) to 255 (saturated).

Type (int, int, int)

`Sprite.event_types = ['on_animation_end']`

Sprite.group

Parent graphics group.

The sprite can change its rendering group, however this can be an expensive operation.

Type *Group*

Sprite.height

Scaled height of the sprite.

Read-only. Invariant under rotation.

Type int

Sprite.image

Image or animation to display.

Type *AbstractImage* or *Animation*

Sprite.opacity

Blend opacity.

This property sets the alpha component of the colour of the sprite's vertices. With the default blend mode (see the constructor), this allows the sprite to be drawn with fractional opacity, blending with the background.

An opacity of 255 (the default) has no effect. An opacity of 128 will make the sprite appear translucent.

Type int

Sprite.position

The (x, y) coordinates of the sprite, as a tuple.

Parameters

- **x** (*int*) – X coordinate of the sprite.
- **y** (*int*) – Y coordinate of the sprite.

Sprite.rotation

Clockwise rotation of the sprite, in degrees.

The sprite image will be rotated about its image's (anchor_x, anchor_y) position.

Type float

Sprite.scale

Scaling factor.

A scaling factor of 1 (the default) has no effect. A scale of 2 will draw the sprite at twice the native size of its image.

Type float

Sprite.scale_x

Horizontal scaling factor.

A scaling factor of 1 (the default) has no effect. A scale of 2 will draw the sprite at twice the native width of its image.

Type float

Sprite.scale_y

Vertical scaling factor.

A scaling factor of 1 (the default) has no effect. A scale of 2 will draw the sprite at twice the native height of its image.

Type float

Sprite.visible

True if the sprite will be drawn.

Type bool

Sprite.width

Scaled width of the sprite.

Read-only. Invariant under rotation.

Type int

Sprite.x

X coordinate of the sprite.

Type int

Sprite.y

Y coordinate of the sprite.

Type int

Inherited members**Methods****Sprite.register_event_type** (*name*)

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters **name** (*str*) – Name of the event to register.

pyglet.graphics.Group

pyglet.sprite.SpriteGroup

**SpriteGroup Class****class SpriteGroup** (*texture, blend_src, blend_dest, parent=None*)

Shared sprite rendering group.

The group is automatically coalesced with other sprite groups sharing the same parent group, texture and blend parameters.

Variables

absolute_import = **_Feature((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)**

compat_platform = **'linux'**

`str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str`

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of `object.__str__()` (if defined) or `repr(object)`. encoding defaults to `sys.getdefaultencoding()`. errors defaults to `'strict'`.

print_function = **_Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)**

Notes

Defined

- clock
- event
- gl
- glctx_arb
- glu
- graphics
- image
- lib
- lib_glx
- math
- sys
- warnings

pyglet.text

Text formatting, layout and display.

This module provides classes for loading styled documents from text files, HTML files and a pyglet-specific markup format. Documents can be styled with multiple fonts, colours, styles, text sizes, margins, paragraph alignments, and so on.

Using the layout classes, documents can be laid out on a single line or word-wrapped to fit a rectangle. A layout can then be efficiently drawn in a window or updated incrementally (for example, to support interactive text editing).

The label classes provide a simple interface for the common case where an application simply needs to display some text in a window.

A plain text label can be created with:

```
label = pyglet.text.Label('Hello, world',
                           font_name='Times New Roman',
                           font_size=36,
                           x=10, y=10)
```

Alternatively, a styled text label using HTML can be created with:

```
label = pyglet.text.HTMLLabel('<b>Hello</b>, <i>world</i>',
                              x=10, y=10)
```

Either label can then be drawn at any time with:

```
label.draw()
```

For details on the subset of HTML supported, see *pyglet.text.formats.html*.

Refer to the Programming Guide for advanced usage of the document and layout classes, including interactive editing, embedding objects within documents and creating scrollable layouts.

Note: Since pyglet 1.1

Modules

<i>caret</i>	Provides keyboard and mouse editing procedures for text layout.
<i>document</i>	Formatted and unformatted document interfaces used by text layout.
<i>formats</i>	Document formats.
<i>layout</i>	Render simple text and formatted documents efficiently.
<i>runlist</i>	Run list encoding utilities.

pyglet.text.caret Provides keyboard and mouse editing procedures for text layout.

Example usage:

```
from pyglet import window
from pyglet.text import layout, caret

my_window = window.Window(...)
my_layout = layout.IncrementalTextLayout(...)
my_caret = caret.Caret(my_layout)
my_window.push_handlers(my_caret)
```

Note: Since pyglet 1.1

<i>Caret</i>	Visible text insertion marker for <i>pyglet.text.layout.IncrementalTextLayout</i> .
--------------	---

Classes

pyglet.text.caret.Caret

Caret Class

class **Caret** (*layout*, *batch=None*, *color=(0, 0, 0)*)

Visible text insertion marker for *pyglet.text.layout.IncrementalTextLayout*.

The caret is drawn as a single vertical bar at the document *position* on a text layout object. If *mark* is not *None*, it gives the unmoving end of the current text selection. The visible text selection on the layout is updated along with *mark* and *position*.

By default the layout's graphics batch is used, so the caret does not need to be drawn explicitly. Even if a different graphics batch is supplied, the caret will be correctly positioned and clipped within the layout.

Updates to the document (and so the layout) are automatically propagated to the caret.

The caret object can be pushed onto a window event handler stack with *Window.push_handlers*. The caret will respond correctly to keyboard, text, mouse and activation events, including double- and triple-clicks. If the text layout is being used alongside other graphical widgets, a GUI toolkit will be needed to delegate keyboard and mouse events to the appropriate widget. pyglet does not provide such a toolkit at this stage.

Attributes:

<i>PERIOD</i>	Blink period, in seconds.
<i>SCROLL_INCREMENT</i>	Pixels to scroll viewport per mouse scroll wheel movement.
<i>color</i>	Caret color.
<i>line</i>	Index of line containing the caret's position.
<i>mark</i>	Position of immovable end of text selection within document.
<i>position</i>	Position of caret within document.
<i>visible</i>	Caret visibility.

Attributes

Caret.PERIOD = 0.5

Blink period, in seconds.

Caret.SCROLL_INCREMENT = 16

Pixels to scroll viewport per mouse scroll wheel movement. Defaults to 12pt at 96dpi.

Caret.color

Caret color.

The default caret color is [0, 0, 0] (black). Each RGB color component is in the range 0 to 255.

Type (int, int, int)

Caret.line

Index of line containing the caret's position.

When set, *position* is modified to place the caret on requested line while maintaining the closest possible X offset.

Type int

Caret.mark

Position of immovable end of text selection within document.

An interactive text selection is determined by its immovable end (the caret's position when a mouse drag begins) and the caret's position, which moves interactively by mouse and keyboard input.

This property is *None* when there is no selection.

Type int

`Caret.position`

Position of caret within document.

Type int

`Caret.visible`

Caret visibility.

The caret may be hidden despite this property due to the periodic blinking or by `on_deactivate` if the event handler is attached to a window.

Type bool

Defined

Notes

- clock
- event
- key
- re
- time

`pyglet.text.document` Formatted and unformatted document interfaces used by text layout.

Abstract representation Styled text in pyglet is represented by one of the *AbstractDocument* classes, which manage the state representation of text and style independently of how it is loaded or rendered.

A document consists of the document text (a Unicode string) and a set of named style ranges. For example, consider the following (artificial) example:

```

0      5      10      15      20
The cat sat on the mat.
+++++++          ++++++      "bold"
                        ++++++      "italic"
```

If this example were to be rendered, “The cat” and “the mat” would be in bold, and “on the” in italics. Note that the second “the” is both bold and italic.

The document styles recorded for this example would be `"bold"` over ranges (0-7, 15-22) and `"italic"` over range (12-18). Overlapping styles are permitted; unlike HTML and other structured markup, the ranges need not be nested.

The document has no knowledge of the semantics of `"bold"` or `"italic"`, it stores only the style names. The pyglet layout classes give meaning to these style names in the way they are rendered; but you are also free to invent your own style names (which will be ignored by the layout classes). This can be useful to tag areas of interest in a document, or maintain references back to the source material.

As well as text, the document can contain arbitrary elements represented by *InlineElement*. An inline element behaves like a single character in the documented, but can be rendered by the application.

Paragraph breaks Paragraph breaks are marked with a “newline” character (U+0010). The Unicode paragraph break (U+2029) can also be used.

Line breaks (U+2028) can be used to force a line break within a paragraph.

See Unicode recommendation UTR #13 for more information: <http://unicode.org/reports/tr13/tr13-5.html>.

Document classes Any class implementing *AbstractDocument* provides an interface to a document model as described above. In theory a structured document such as HTML or XML could export this model, though the classes provided by pyglet implement only unstructured documents.

The *UnformattedDocument* class assumes any styles set are set over the entire document. So, regardless of the range specified when setting a "bold" style attribute, for example, the entire document will receive that style.

The *FormattedDocument* class implements the document model directly, using the *RunList* class to represent style runs efficiently.

Style attributes The following character style attribute names are recognised by pyglet:

font_name Font family name, as given to *pyglet.font.load*.

font_size Font size, in points.

bold Boolean.

italic Boolean.

underline 4-tuple of ints in range (0, 255) giving RGBA underline color, or *None* (default) for no underline.

kerining Additional space to insert between glyphs, in points. Defaults to 0.

baseline Offset of glyph baseline from line baseline, in points. Positive values give a superscript, negative values give a subscript. Defaults to 0.

color 4-tuple of ints in range (0, 255) giving RGBA text color

background_color 4-tuple of ints in range (0, 255) giving RGBA text background color; or *None* for no background fill.

The following paragraph style attribute names are recognised by pyglet. Note that paragraph styles are handled differently from character styles by the document: it is the application's responsibility to set the style over an entire paragraph, otherwise results are undefined.

align *left* (default), *center* or *right*.

indent Additional horizontal space to insert before the first

leading Additional space to insert between consecutive lines within a paragraph, in points. Defaults to 0.

line_spacing Distance between consecutive baselines in a paragraph, in points. Defaults to *None*, which automatically calculates the tightest line spacing for each line based on the font ascent and descent.

margin_left Left paragraph margin, in pixels.

margin_right Right paragraph margin, in pixels.

margin_top Margin above paragraph, in pixels.

margin_bottom Margin below paragraph, in pixels. Adjacent margins do not collapse.

tab_stops List of horizontal tab stops, in pixels, measured from the left edge of the text layout. Defaults to the empty list. When the tab stops are exhausted, they implicitly continue at 50 pixel intervals.

wrap Boolean. If *True* (the default), text wraps within the width of the layout.

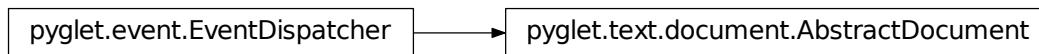
Other attributes can be used to store additional style information within the document; it will be ignored by the built-in text classes.

All style attributes (including those not present in a document) default to *None* (including the so-called "boolean" styles listed above). The meaning of a *None* style is style- and application-dependent.

Note: Since pyglet 1.1

<i>AbstractDocument</i>	Abstract document interface used by all <i>pyglet.text</i> classes.
<i>FormattedDocument</i>	Simple implementation of a document that maintains text formatting.
<i>InlineElement</i>	Arbitrary inline element positioned within a formatted document.
<i>UnformattedDocument</i>	A document having uniform style over all text.

Classes



AbstractDocument Class

class **AbstractDocument** (*text*='')

Abstract document interface used by all *pyglet.text* classes.

This class can be overridden to interface pyglet with a third-party document format. It may be easier to implement the document format in terms of one of the supplied concrete classes *FormattedDocument* or *UnformattedDocument*.

Methods:

Attributes:

<i>event_types</i>	
<i>text</i>	Document text.

Attributes

`AbstractDocument.event_types = ['on_insert_text', 'on_delete_text', 'on_style_text']`

`AbstractDocument.text`

Document text.

For efficient incremental updates, use the *insert_text* and *delete_text* methods instead of replacing this property.

Type str

Inherited members

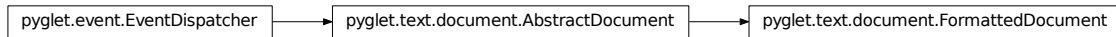
Methods

`AbstractDocument.register_event_type(name)`

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters `name` (*str*) – Name of the event to register.



FormattedDocument Class

`class FormattedDocument(text='')`

Simple implementation of a document that maintains text formatting.

Changes to text style are applied according to the description in *AbstractDocument*. All styles default to `None`.

Methods:

Attributes:

<code>event_types</code>	
<code>text</code>	Document text.

Inherited members

Methods

`FormattedDocument.register_event_type(name)`

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters `name` (*str*) – Name of the event to register.

Attributes

`FormattedDocument.event_types = ['on_insert_text', 'on_delete_text', 'on_style_text']`

`FormattedDocument.text`

Document text.

For efficient incremental updates, use the *insert_text* and *delete_text* methods instead of replacing this property.

Type str

pyglet.text.document.InlineElement

***InlineElement* Class**

class `InlineElement` (*ascent*, *descent*, *advance*)

Arbitrary inline element positioned within a formatted document.

Elements behave like a single glyph in the document. They are measured by their horizontal advance, ascent above the baseline, and descent below the baseline.

The pyglet layout classes reserve space in the layout for elements and call the element's methods to ensure they are rendered at the appropriate position.

If the size of a element (any of the *advance*, *ascent*, or *descent* instance variables) is modified it is the application's responsibility to trigger a reflow of the appropriate area in the affected layouts. This can be done by forcing a style change over the element's position.

Variables

- ***ascent*** – Ascent of the element above the baseline, in pixels.
- ***descent*** – Descent of the element below the baseline, in pixels. Typically negative.
- ***advance*** – Width of the element, in pixels.

Attributes:

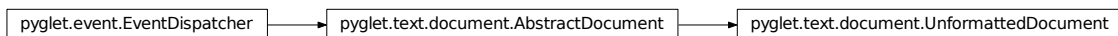
position Position of the element within the document.

Attributes

`InlineElement.position`

Position of the element within the document. Read-only.

Type int



***UnformattedDocument* Class**

class `UnformattedDocument` (*text*='')

A document having uniform style over all text.

Changes to the style of text within the document affects the entire document. For convenience, the `position` parameters of the style methods may therefore be omitted.

Methods:

Attributes:

<code>event_types</code>	
<code>text</code>	Document text.

Inherited members

Methods

`UnformattedDocument.register_event_type(name)`

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters `name` (*str*) – Name of the event to register.

Attributes

`UnformattedDocument.event_types = ['on_insert_text', 'on_delete_text', 'on_style_text']`

`UnformattedDocument.text`

Document text.

For efficient incremental updates, use the *insert_text* and *delete_text* methods instead of replacing this property.

Type *str*

Variables

`STYLE_INDETERMINATE = 'indeterminate'`

The style attribute takes on multiple values in the document.

	Defined
Notes	<ul style="list-style-type: none"> • <code>event</code> • <code>re</code> • <code>runlist</code> • <code>sys</code>

`pyglet.text.formats` Document formats.

Note: Since pyglet 1.1

<i>attributed</i>	Extensible attributed text format for representing pyglet formatted documents.
<i>html</i>	Decode HTML into attributed text.
<i>plaintext</i>	Plain text decoder.
<i>structured</i>	Base class for structured (hierarchical) document formats.

Modules

pyglet.text.formats.attributed Extensible attributed text format for representing pyglet formatted documents.

AttributedTextDecoder

Classes



AttributedTextDecoder Class

class **AttributedTextDecoder**

Defined

Notes

- operator
- parser
- pyglet
- re
- token

pyglet.text.formats.html Decode HTML into attributed text.

A subset of HTML 4.01 Transitional is implemented. The following elements are supported fully:

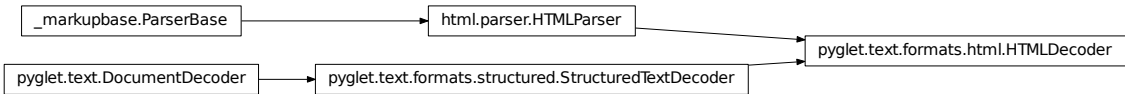
B BLOCKQUOTE BR CENTER CODE DD DIR DL EM FONT H1 H2 H3 H4 H5 H6 I IMG KBD LI MENU OL P PRE Q SAMP STRONG SUB SUP TT U UL VAR

The mark (bullet or number) of a list item is separated from the body of the list item with a tab, as the pyglet document model does not allow out-of-stream text. This means lists display as expected, but behave a little oddly if edited.

No CSS styling is supported.

HTMLDecoder Decoder for HTML documents.

Classes



HTMLDecoder Class

class HTMLDecoder (*, *convert_charrefs=True*)
Decoder for HTML documents.

Attributes:

CDATA_CONTENT_ELEMENTS	
<i>default_style</i>	Default style attributes for unstyled text in the HTML document.
<i>font_sizes</i>	Map HTML font sizes to actual font sizes, in points.

Attributes

`HTMLDecoder.default_style` = {'margin_bottom': '12pt', 'font_size': 12, 'font_name': 'Times New Roman'}
Default style attributes for unstyled text in the HTML document.

Type dict

`HTMLDecoder.font_sizes` = {1: 8, 2: 10, 3: 12, 4: 14, 5: 18, 6: 24, 7: 48}
Map HTML font sizes to actual font sizes, in points.

Type dict

Inherited members

Attributes

`HTMLDecoder.CDATA_CONTENT_ELEMENTS` = ('script', 'style')

Defined

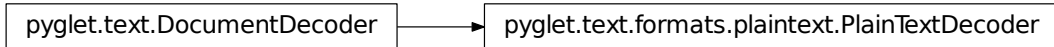
Notes

- entities
- pyglet
- re
- structured

`pyglet.text.formats.plaintext` Plain text decoder.

PlainTextDecoder

Classes



PlainTextDecoder Class

class **PlainTextDecoder**

Defined
<div>Notes</div> <ul style="list-style-type: none"> • pyglet

pyglet.text.formats.structured Base class for structured (hierarchical) document formats.

ImageElement

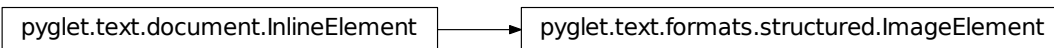
ListBuilder

OrderedListBuilder

StructuredTextDecoder

UnorderedListBuilder

Classes



ImageElement Class

class **ImageElement** (*image*, *width=None*, *height=None*)

Attributes:

position Position of the element within the document.

Inherited members

Attributes

`ImageElement.position`

Position of the element within the document. Read-only.

Type `int`

pyglet.text.formats.structured.ListBuilder

ListBuilder Class

class `ListBuilder`

pyglet.text.formats.structured.ListBuilder

pyglet.text.formats.structured.OrderedListBuilder

OrderedListBuilder Class

class `OrderedListBuilder` (*start*, *format*)

Attributes:

format_re

Attributes

`OrderedListBuilder.format_re` = `re.compile('(.*?)([!a-zA-Z])(.*)')`

pyglet.text.DocumentDecoder

pyglet.text.formats.structured.StructuredTextDecoder

StructuredTextDecoder Class

class `StructuredTextDecoder`

pyglet.text.formats.structured.ListBuilder

pyglet.text.formats.structured.UnorderedListBuilder

UnorderedListBuilder Class**class UnorderedListBuilder** (*mark*)**Variables****division** = `_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)`**Defined****Notes**

- pyglet
- re

pyglet.text.layout Render simple text and formatted documents efficiently.

Three layout classes are provided:

TextLayout The entire document is laid out before it is rendered. The layout will be grouped with other layouts in the same batch (allowing for efficient rendering of multiple layouts).

Any change to the layout or document, and even querying some properties, will cause the entire document to be laid out again.

ScrollableTextLayout Based on *TextLayout*.A separate group is used for layout which crops the contents of the layout to the layout rectangle. Additionally, the contents of the layout can be “scrolled” within that rectangle with the `view_x` and `view_y` properties.***IncrementalTextLayout*** Based on *ScrollableTextLayout*.

When the layout or document are modified, only the affected regions are laid out again. This permits efficient interactive editing and styling of text.

Only the visible portion of the layout is actually rendered; as the viewport is scrolled additional sections are rendered and discarded as required. This permits efficient viewing and editing of large documents.

Additionally, this class provides methods for locating the position of a caret in the document, and for displaying interactive text selections.

All three layout classes can be used with either *UnformattedDocument* or *FormattedDocument*, and can be either single-line or multiline. The combinations of these options effectively provides 12 different text display possibilities.**Style attributes** The following character style attribute names are recognised by the layout classes. Data types and units are as specified.

Where an attribute is marked “as a distance” the value is assumed to be in pixels if given as an int or float, otherwise a string of the form “0u” is required, where 0 is the distance and u is the unit; one of “px” (pixels), “pt” (points), “pc” (picas), “cm” (centimeters), “mm” (millimeters) or “in” (inches). For example, “14pt” is the distance covering 14 points, which at the default DPI of 96 is 18 pixels.

font_name Font family name, as given to *pyglet.font.load*.

font_size Font size, in points.

bold Boolean.

italic Boolean.

underline 4-tuple of ints in range (0, 255) giving RGBA underline color, or *None* (default) for no underline.

Kerning Additional space to insert between glyphs, as a distance. Defaults to 0.

baseline Offset of glyph baseline from line baseline, as a distance. Positive values give a superscript, negative values give a subscript. Defaults to 0.

color 4-tuple of ints in range (0, 255) giving RGBA text color

background_color 4-tuple of ints in range (0, 255) giving RGBA text background color; or *None* for no background fill.

The following paragraph style attribute names are recognised. Note that paragraph styles are handled no differently from character styles by the document: it is the application’s responsibility to set the style over an entire paragraph, otherwise results are undefined.

align left (default), center or right.

indent Additional horizontal space to insert before the first glyph of the first line of a paragraph, as a distance.

leading Additional space to insert between consecutive lines within a paragraph, as a distance. Defaults to 0.

line_spacing Distance between consecutive baselines in a paragraph, as a distance. Defaults to *None*, which automatically calculates the tightest line spacing for each line based on the font ascent and descent.

margin_left Left paragraph margin, as a distance.

margin_right Right paragraph margin, as a distance.

margin_top Margin above paragraph, as a distance.

margin_bottom Margin below paragraph, as a distance. Adjacent margins do not collapse.

tab_stops List of horizontal tab stops, as distances, measured from the left edge of the text layout. Defaults to the empty list. When the tab stops are exhausted, they implicitly continue at 50 pixel intervals.

wrap char, word, True (default) or False. The boundaries at which to wrap text to prevent it overflowing a line. With *char*, the line wraps anywhere in the text; with *word* or *True*, the line wraps at appropriate boundaries between words; with *False* the line does not wrap, and may overflow the layout width. *char* and *word* styles are since pyglet 1.2.

Other attributes can be used to store additional style information within the document; they will be ignored by the built-in text classes.

Note: Since pyglet 1.1

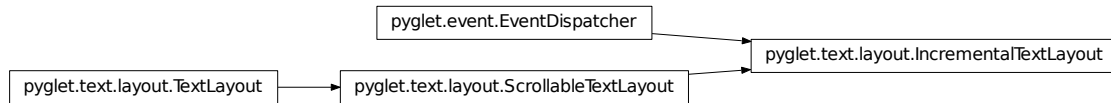
<i>IncrementalTextLayout</i>	Displayed text suitable for interactive editing and/or scrolling large documents
<i>ScrollableTextLayout</i>	Display text in a scrollable viewport.
<i>ScrollableTextLayoutGroup</i>	Top-level rendering group for <i>ScrollableTextLayout</i> .
<i>TextLayout</i>	Lay out and display documents.
<i>TextLayoutForegroundDecorationGroup</i>	Rendering group for decorative elements (e.g., glyph underlines) in all text layouts.
<i>TextLayoutForegroundGroup</i>	Rendering group for foreground elements (glyphs) in all text layouts.

Continued on next page

Table 2.201 – continued from previous page

<i>TextLayoutGroup</i>	Top-level rendering group for <i>TextLayout</i> .
<i>TextLayoutTextureGroup</i>	Rendering group for a glyph texture in all text layouts.

Classes



IncrementalTextLayout Class

class `IncrementalTextLayout` (*document, width, height, multiline=False, dpi=None, batch=None, group=None, wrap_lines=True*)

Displayed text suitable for interactive editing and/or scrolling large documents.

Unlike *TextLayout* and *ScrollableTextLayout*, this class generates vertex lists only for lines of text that are visible. As the document is scrolled, vertex lists are deleted and created as appropriate to keep video memory usage to a minimum and improve rendering speed.

Changes to the document are quickly reflected in this layout, as only the affected line(s) are reflowed. Use *begin_update* and *end_update* to further reduce the amount of processing required.

The layout can also display a text selection (text with a different background color). The *Caret* class implements a visible text cursor and provides event handlers for scrolling, selecting and editing text in an incremental text layout.

Methods:

Attributes:

<code>anchor_x</code>	
<code>anchor_y</code>	
<code>content_valign</code>	Vertical alignment of content within larger layout box.
<code>document</code>	
<code>dpi</code>	Get DPI used by this layout.
<code>event_types</code>	
<code>foreground_decoration_group</code>	
<code>foreground_group</code>	
<code>height</code>	
<code>multiline</code>	
<code>selection_background_color</code>	Background color of active selection.
<code>selection_color</code>	Text color of active selection.
<code>selection_end</code>	End position of the active selection (exclusive).
<code>selection_start</code>	Starting position of the active selection.
<code>top_group</code>	
<code>view_x</code>	Horizontal scroll offset.
<code>view_y</code>	

Continued on next page

Table 2.203 – continued from previous page

<i>width</i>
<i>x</i>
<i>y</i>

Attributes

`IncrementalTextLayout.event_types = ['on_layout_update']`

`IncrementalTextLayout.height`

`IncrementalTextLayout.multiline`

`IncrementalTextLayout.selection_background_color`

Background color of active selection.

The color is an RGBA tuple with components in range [0, 255].

Type (int, int, int, int)

`IncrementalTextLayout.selection_color`

Text color of active selection.

The color is an RGBA tuple with components in range [0, 255].

Type (int, int, int, int)

`IncrementalTextLayout.selection_end`

End position of the active selection (exclusive).

See *set_selection*

Type int

`IncrementalTextLayout.selection_start`

Starting position of the active selection.

See *set_selection*

Type int

`IncrementalTextLayout.view_y`

`IncrementalTextLayout.width`

Inherited members

Methods

`IncrementalTextLayout.register_event_type(name)`

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters *name* (*str*) – Name of the event to register.

Attributes

`IncrementalTextLayout.anchor_x`

`IncrementalTextLayout.anchor_y`

`IncrementalTextLayout.background_group = OrderedGroup(0)`

`IncrementalTextLayout.content_valign`

Vertical alignment of content within larger layout box.

This property determines how content is positioned within the layout box when `content_height` is less than `height`. It is one of the enumerants:

top (default) Content is aligned to the top of the layout box.

center Content is centered vertically within the layout box.

bottom Content is aligned to the bottom of the layout box.

This property has no effect when `content_height` is greater than `height` (in which case the content is aligned to the top) or when `height` is `None` (in which case there is no vertical layout box dimension).

Type str

`IncrementalTextLayout.document`

`IncrementalTextLayout.dpi`

Get DPI used by this layout.

Read-only.

Type float

`IncrementalTextLayout.foreground_decoration_group = TextLayoutForegroundDecorationGroup(2)`

`IncrementalTextLayout.foreground_group = TextLayoutForegroundGroup(1)`

`IncrementalTextLayout.top_group = <pyglet.text.layout.TextLayoutGroup object>`

`IncrementalTextLayout.view_x`

Horizontal scroll offset.

The initial value is 0, and the left edge of the text will touch the left side of the layout bounds. A positive value causes the text to “scroll” to the right. Values are automatically clipped into the range `[0, content_width - width]`

Type int

`IncrementalTextLayout.x`

`IncrementalTextLayout.y`

`pyglet.text.layout.TextLayout`

`pyglet.text.layout.ScrollableTextLayout`

ScrollableTextLayout Class

```
class ScrollableTextLayout (document, width, height, multiline=False, dpi=None, batch=None,
                           group=None, wrap_lines=True)
```

Display text in a scrollable viewport.

This class does not display a scrollbar or handle scroll events; it merely clips the text that would be drawn in *TextLayout* to the bounds of the layout given by *x*, *y*, *width* and *height*; and offsets the text by a scroll offset.

Use *view_x* and *view_y* to scroll the text within the viewport.

Attributes:

<i>anchor_x</i>	
<i>anchor_y</i>	
<i>content_valign</i>	Vertical alignment of content within larger layout box.
<i>document</i>	
<i>dpi</i>	Get DPI used by this layout.
<i>foreground_decoration_group</i>	
<i>foreground_group</i>	
<i>height</i>	
<i>multiline</i>	Set if multiline layout is enabled.
<i>top_group</i>	
<i>view_x</i>	Horizontal scroll offset.
<i>view_y</i>	Vertical scroll offset.
<i>width</i>	
<i>x</i>	
<i>y</i>	

Attributes

`ScrollableTextLayout.anchor_x`

`ScrollableTextLayout.anchor_y`

`ScrollableTextLayout.height`

`ScrollableTextLayout.view_x`

Horizontal scroll offset.

The initial value is 0, and the left edge of the text will touch the left side of the layout bounds. A positive value causes the text to “scroll” to the right. Values are automatically clipped into the range `[0, content_width - width]`

Type `int`

`ScrollableTextLayout.view_y`

Vertical scroll offset.

The initial value is 0, and the top of the text will touch the top of the layout bounds (unless the content height is less than the layout height, in which case *content_valign* is used).

A negative value causes the text to “scroll” upwards. Values outside of the range `[height - content_height, 0]` are automatically clipped in range.

Type `int`

`ScrollableTextLayout.width`

`ScrollableTextLayout.x`

`ScrollableTextLayout.y`

Inherited members

Attributes

`ScrollableTextLayout.background_group = OrderedGroup(0)`

`ScrollableTextLayout.content_valign`

Vertical alignment of content within larger layout box.

This property determines how content is positioned within the layout box when `content_height` is less than `height`. It is one of the enumerants:

top (default) Content is aligned to the top of the layout box.

center Content is centered vertically within the layout box.

bottom Content is aligned to the bottom of the layout box.

This property has no effect when `content_height` is greater than `height` (in which case the content is aligned to the top) or when `height` is `None` (in which case there is no vertical layout box dimension).

Type str

`ScrollableTextLayout.document`

`ScrollableTextLayout.dpi`

Get DPI used by this layout.

Read-only.

Type float

`ScrollableTextLayout.foreground_decoration_group = TextLayoutForegroundDecorationGroup(2)`

`ScrollableTextLayout.foreground_group = TextLayoutForegroundGroup(1)`

`ScrollableTextLayout.multiline`

Set if multiline layout is enabled.

If multiline is False, newline and paragraph characters are ignored and text is not word-wrapped. If True, the text is word-wrapped only if the `wrap_lines` is True.

Type bool

`ScrollableTextLayout.top_group = <pyglet.text.layout.TextLayoutGroup object>`

pyglet.graphics.Group

pyglet.text.layout.ScrollableTextLayout

ScrollableTextLayoutGroup Class

class ScrollableTextLayoutGroup (*parent=None*)

Top-level rendering group for *ScrollableTextLayout*.

The group maintains internal state for setting the clipping planes and view transform for scrolling. Because the group has internal state specific to the text layout, the group is never shared.

Attributes:

<code>height</code>	Height of the text layout.
<code>left</code>	Left edge of the text layout.
<code>top</code>	Top edge of the text layout (measured from the bottom of the graphics viewport).
<code>translate_x</code>	
<code>translate_y</code>	
<code>view_x</code>	Horizontal scroll offset.
<code>view_y</code>	Vertical scroll offset.
<code>width</code>	Width of the text layout.

Attributes

`ScrollableTextLayoutGroup.height`
Height of the text layout.

Type int

`ScrollableTextLayoutGroup.left`
Left edge of the text layout.

Type int

`ScrollableTextLayoutGroup.top`
Top edge of the text layout (measured from the bottom of the graphics viewport).

Type int

`ScrollableTextLayoutGroup.translate_x = 0`

`ScrollableTextLayoutGroup.translate_y = 0`

`ScrollableTextLayoutGroup.view_x`
Horizontal scroll offset.

Type int

`ScrollableTextLayoutGroup.view_y`
Vertical scroll offset.

Type int

`ScrollableTextLayoutGroup.width`
Width of the text layout.

Type int

pyglet.text.layout.TextLayout

***TextLayout* Class**

class TextLayout (*document*, *width=None*, *height=None*, *multiline=False*, *dpi=None*, *batch=None*, *group=None*, *wrap_lines=True*)

Lay out and display documents.

This class is intended for displaying documents that do not change regularly – any change will cost some time to lay out the complete document again and regenerate all vertex lists.

The benefit of this class is that texture state is shared between all layouts of this class. The time to draw one *TextLayout* may be roughly the same as the time to draw one *IncrementalTextLayout*; but drawing ten *TextLayout* objects in one batch is much faster than drawing ten incremental or scrollable text layouts.

Label and *HTMLLabel* provide a convenient interface to this class.

Variables

- **content_width** – Calculated width of the text in the layout. This may overflow the desired width if word-wrapping failed.
- **content_height** – Calculated height of the text in the layout.
- **top_group** – Top-level rendering group.
- **background_group** – Rendering group for background color.
- **foreground_group** – Rendering group for glyphs.
- **foreground_decoration_group** – Rendering group for glyph underlines.

Attributes:

<i>anchor_x</i>	Horizontal anchor alignment.
<i>anchor_y</i>	Vertical anchor alignment.
<i>content_valign</i>	Vertical alignment of content within larger layout box.
<i>document</i>	
<i>dpi</i>	Get DPI used by this layout.
<i>foreground_decoration_group</i>	
<i>foreground_group</i>	
<i>height</i>	Height of the layout.
<i>multiline</i>	Set if multiline layout is enabled.
<i>top_group</i>	
<i>width</i>	Width of the layout.
<i>x</i>	X coordinate of the layout.
<i>y</i>	Y coordinate of the layout.

Attributes

`TextLayout.anchor_x`

Horizontal anchor alignment.

This property determines the meaning of the *x* coordinate. It is one of the enumerants:

"left" (default) The X coordinate gives the position of the left edge of the layout.

"center" The X coordinate gives the position of the center of the layout.

"right" The X coordinate gives the position of the right edge of the layout.

For the purposes of calculating the position resulting from this alignment, the width of the layout is taken to be *width* if *multiline* is True and *wrap_lines* is True, otherwise *content_width*.

Type str

`TextLayout.anchor_y`

Vertical anchor alignment.

This property determines the meaning of the y coordinate. It is one of the enumerants:

"top" The Y coordinate gives the position of the top edge of the layout.

"center" The Y coordinate gives the position of the center of the layout.

"baseline" The Y coordinate gives the position of the baseline of the first line of text in the layout.

"bottom" (default) The Y coordinate gives the position of the bottom edge of the layout.

For the purposes of calculating the position resulting from this alignment, the height of the layout is taken to be the smaller of *height* and *content_height*.

See also *content_valign*.

Type str

`TextLayout.background_group = OrderedGroup(0)`

`TextLayout.content_valign`

Vertical alignment of content within larger layout box.

This property determines how content is positioned within the layout box when *content_height* is less than *height*. It is one of the enumerants:

top (default) Content is aligned to the top of the layout box.

center Content is centered vertically within the layout box.

bottom Content is aligned to the bottom of the layout box.

This property has no effect when *content_height* is greater than *height* (in which case the content is aligned to the top) or when *height* is None (in which case there is no vertical layout box dimension).

Type str

`TextLayout.document`

`TextLayout.dpi`

Get DPI used by this layout.

Read-only.

Type float

`TextLayout.foreground_decoration_group = TextLayoutForegroundDecorationGroup(2)`

`TextLayout.foreground_group = TextLayoutForegroundGroup(1)`

`TextLayout.height`

Height of the layout.

Type int

`TextLayout.multiline`

Set if multiline layout is enabled.

If multiline is False, newline and paragraph characters are ignored and text is not word-wrapped. If True, the text is word-wrapped only if the *wrap_lines* is True.

Type bool

`TextLayout.top_group = <pyglet.text.layout.TextLayoutGroup object>`

`TextLayout.width`

Width of the layout.

This property has no effect if *multiline* is False or *wrap_lines* is False.

Type int

`TextLayout.x`

X coordinate of the layout.

See also *anchor_x*.

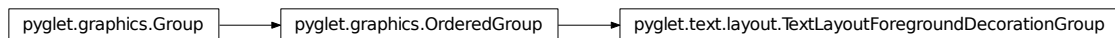
Type int

`TextLayout.y`

Y coordinate of the layout.

See also *anchor_y*.

Type int

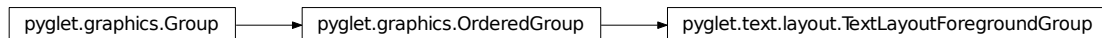


***TextLayoutForegroundDecorationGroup* Class**

class `TextLayoutForegroundDecorationGroup` (*order*, *parent=None*)

Rendering group for decorative elements (e.g., glyph underlines) in all text layouts.

The group disables `GL_TEXTURE_2D`.

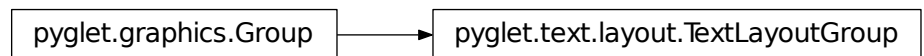


***TextLayoutForegroundGroup* Class**

class `TextLayoutForegroundGroup` (*order*, *parent=None*)

Rendering group for foreground elements (glyphs) in all text layouts.

The group enables `GL_TEXTURE_2D`.

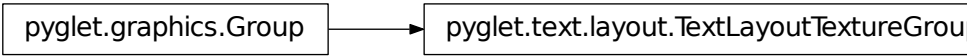


***TextLayoutGroup* Class**

class `TextLayoutGroup` (*parent=None*)

Top-level rendering group for *TextLayout*.

The blend function is set for glyph rendering (GL_SRC_ALPHA / GL_ONE_MINUS_SRC_ALPHA). The group is shared by all *TextLayout* instances as it has no internal state.



TextLayoutTextureGroup Class

class TextLayoutTextureGroup (*texture, parent*)
Rendering group for a glyph texture in all text layouts.

The group binds its texture to GL_TEXTURE_2D. The group is shared between all other text layout uses of the same texture.

Variables

absolute_import = **_Feature**((2, 5, 0, 'alpha', 1), (3, 0, 0, 'alpha', 0), 16384)
compat_platform = 'linux'
str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

division = **_Feature**((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
print_function = **_Feature**((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)

Defined

- event
- gl
- glext_arb
- glu
- graphics
- lib
- lib_glx
- re
- runlist
- sys

Notes

pyglet.text.runlist Run list encoding utilities.

Note: Since pyglet 1.1

<i>AbstractRunIterator</i>	Range iteration over <i>RunList</i> .
<i>ConstRunIterator</i>	Iterate over a constant value without creating a <i>RunList</i> .
<i>FilteredRunIterator</i>	Iterate over an <i>AbstractRunIterator</i> with filtered values replaced by a default value.
<i>OverriddenRunIterator</i>	Iterator over a <i>RunIterator</i> , with a value temporarily replacing a given range.
<i>RunIterator</i>	
<i>RunList</i>	List of contiguous runs of values.
<i>ZipRunIterator</i>	Iterate over multiple run iterators concurrently.

Classes

pyglet.text.runlist.AbstractRunIterator

AbstractRunIterator Class

class **AbstractRunIterator**

Range iteration over *RunList*.

AbstractRunIterator objects allow any monotonically non-decreasing access of the iteration, including repeated iteration over the same index. Use the `[index]` operator to get the value at a particular index within the document. For example:

```
run_iter = iter(run_list)
value = run_iter[0]
value = run_iter[0]           # non-decreasing access is OK
value = run_iter[15]
value = run_iter[17]
value = run_iter[16]         # this is illegal, the index decreased.
```

Using *AbstractRunIterator* to access increasing indices of the value runs is more efficient than calling *RunList.__getitem__* repeatedly.

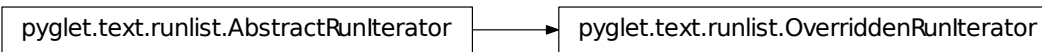
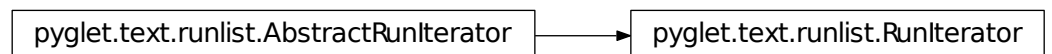
You can also iterate over monotonically non-decreasing ranges over the iteration. For example:

```
run_iter = iter(run_list)
for start, end, value in run_iter.ranges(0, 20):
    pass
for start, end, value in run_iter.ranges(25, 30):
    pass
for start, end, value in run_iter.ranges(30, 40):
    pass
```

Both start and end indices of the slice are required and must be positive.

***ConstRunIterator Class*****class ConstRunIterator** (*length, value*)

Iterate over a constant value without creating a RunList.

***FilteredRunIterator Class*****class FilteredRunIterator** (*base_iterator, filter, default*)Iterate over an *AbstractRunIterator* with filtered values replaced by a default value.***OverriddenRunIterator Class*****class OverriddenRunIterator** (*base_iterator, start, end, value*)Iterator over a *RunIterator*, with a value temporarily replacing a given range.***RunIterator Class*****class RunIterator** (*run_list*)

pyglet.text.runlist.RunList

***RunList* Class**

class RunList (*size, initial*)

List of contiguous runs of values.

A *RunList* is an efficient encoding of a sequence of values. For example, the sequence `aaaabbccccc` is encoded as `(4, a), (2, b), (5, c)`. The class provides methods for modifying and querying the run list without needing to deal with the tricky cases of splitting and merging the run list entries.

Run lists are used to represent formatted character data in pyglet. A separate run list is maintained for each style attribute, for example, bold, italic, font size, and so on. Unless you are overriding the document interfaces, the only interaction with run lists is via *RunIterator*.

The length and ranges of a run list always refer to the character positions in the decoded list. For example, in the above sequence, `set_run(2, 5, 'x')` would change the sequence to `aaxxxbccccc`.

pyglet.text.runlist.AbstractRunIterator

→

pyglet.text.runlist.ZipRunIterator

***ZipRunIterator* Class**

class ZipRunIterator (*range_iterators*)

Iterate over multiple run iterators concurrently.

Classes

<i>DocumentDecoder</i>	Abstract document decoder.
<i>DocumentLabel</i>	Base label class.
<i>HTMLLabel</i>	HTML formatted text label.
<i>Label</i>	Plain text label.

pyglet.text.DocumentDecoder

***DocumentDecoder* Class**

class DocumentDecoder

Abstract document decoder.

pyglet.text.layout.TextLayout

pyglet.text.DocumentLabel

DocumentLabel Class

```
class DocumentLabel (document=None, x=0, y=0, width=None, height=None, anchor_x='left', anchor_y='baseline', multiline=False, dpi=None, batch=None, group=None)
```

Base label class.

A label is a layout that exposes convenience methods for manipulating the associated document.

Attributes:

<code>anchor_x</code>	Horizontal anchor alignment.
<code>anchor_y</code>	Vertical anchor alignment.
<code>bold</code>	Bold font style.
<code>color</code>	Text color.
<code>content_valign</code>	Vertical alignment of content within larger layout box.
<code>document</code>	
<code>dpi</code>	Get DPI used by this layout.
<code>font_name</code>	Font family name.
<code>font_size</code>	Font size, in points.
<code>height</code>	Height of the layout.
<code>italic</code>	Italic font style.
<code>multiline</code>	Set if multiline layout is enabled.
<code>text</code>	The text of the label.
<code>width</code>	Width of the layout.
<code>x</code>	X coordinate of the layout.
<code>y</code>	Y coordinate of the layout.

Attributes

`DocumentLabel.bold`

Bold font style.

Type bool

`DocumentLabel.color`

Text color.

Color is a 4-tuple of RGBA components, each in range [0, 255].

Type (int, int, int, int)

`DocumentLabel.font_name`

Font family name.

The font name, as passed to `pyglet.font.load`. A list of names can optionally be given: the first matching font will be used.

Type str or list

`DocumentLabel.font_size`

Font size, in points.

Type float

`DocumentLabel.italic`

Italic font style.

Type bool

`DocumentLabel.text`

The text of the label.

Type str

Inherited members

Attributes

`DocumentLabel.anchor_x`

Horizontal anchor alignment.

This property determines the meaning of the x coordinate. It is one of the enumerants:

"left" (default) The X coordinate gives the position of the left edge of the layout.

"center" The X coordinate gives the position of the center of the layout.

"right" The X coordinate gives the position of the right edge of the layout.

For the purposes of calculating the position resulting from this alignment, the width of the layout is taken to be *width* if *multiline* is True and *wrap_lines* is True, otherwise *content_width*.

Type str

`DocumentLabel.anchor_y`

Vertical anchor alignment.

This property determines the meaning of the y coordinate. It is one of the enumerants:

"top" The Y coordinate gives the position of the top edge of the layout.

"center" The Y coordinate gives the position of the center of the layout.

"baseline" The Y coordinate gives the position of the baseline of the first line of text in the layout.

"bottom" (default) The Y coordinate gives the position of the bottom edge of the layout.

For the purposes of calculating the position resulting from this alignment, the height of the layout is taken to be the smaller of *height* and *content_height*.

See also *content_valign*.

Type str

`DocumentLabel.background_group = OrderedGroup(0)`

`DocumentLabel.content_valign`

Vertical alignment of content within larger layout box.

This property determines how content is positioned within the layout box when *content_height* is less than *height*. It is one of the enumerants:

top (default) Content is aligned to the top of the layout box.

center Content is centered vertically within the layout box.

bottom Content is aligned to the bottom of the layout box.

This property has no effect when `content_height` is greater than `height` (in which case the content is aligned to the top) or when `height` is `None` (in which case there is no vertical layout box dimension).

Type str

`DocumentLabel.document`

`DocumentLabel.dpi`

Get DPI used by this layout.

Read-only.

Type float

`DocumentLabel.foreground_decoration_group = TextLayoutForegroundDecorationGroup(2)`

`DocumentLabel.foreground_group = TextLayoutForegroundGroup(1)`

`DocumentLabel.height`

Height of the layout.

Type int

`DocumentLabel.multiline`

Set if multiline layout is enabled.

If `multiline` is `False`, newline and paragraph characters are ignored and text is not word-wrapped. If `True`, the text is word-wrapped only if the `wrap_lines` is `True`.

Type bool

`DocumentLabel.top_group = <pyglet.text.layout.TextLayoutGroup object>`

`DocumentLabel.width`

Width of the layout.

This property has no effect if `multiline` is `False` or `wrap_lines` is `False`.

Type int

`DocumentLabel.x`

X coordinate of the layout.

See also `anchor_x`.

Type int

`DocumentLabel.y`

Y coordinate of the layout.

See also `anchor_y`.

Type int



HTMLLabel Class

class HTMLLabel (*text='', location=None, x=0, y=0, width=None, height=None, anchor_x='left', anchor_y='baseline', multiline=False, dpi=None, batch=None, group=None*)
HTML formatted text label.

A subset of HTML 4.01 is supported. See *pyglet.text.formats.html* for details.

Attributes:

<code>anchor_x</code>	Horizontal anchor alignment.
<code>anchor_y</code>	Vertical anchor alignment.
<code>bold</code>	Bold font style.
<code>color</code>	Text color.
<code>content_valign</code>	Vertical alignment of content within larger layout box.
<code>document</code>	
<code>dpi</code>	Get DPI used by this layout.
<code>font_name</code>	Font family name.
<code>font_size</code>	Font size, in points.
<code>height</code>	Height of the layout.
<code>italic</code>	Italic font style.
<code>multiline</code>	Set if multiline layout is enabled.
<code>text</code>	HTML formatted text of the label.
<code>width</code>	Width of the layout.
<code>x</code>	X coordinate of the layout.
<code>y</code>	Y coordinate of the layout.

Attributes

`HTMLLabel.text`

HTML formatted text of the label.

Type str

Inherited members

Attributes

`HTMLLabel.anchor_x`

Horizontal anchor alignment.

This property determines the meaning of the *x* coordinate. It is one of the enumerants:

"left" (default) The X coordinate gives the position of the left edge of the layout.

"center" The X coordinate gives the position of the center of the layout.

"right" The X coordinate gives the position of the right edge of the layout.

For the purposes of calculating the position resulting from this alignment, the width of the layout is taken to be *width* if *multiline* is True and *wrap_lines* is True, otherwise *content_width*.

Type str

HTMLLabel.**anchor_y**

Vertical anchor alignment.

This property determines the meaning of the y coordinate. It is one of the enumerants:

"top" The Y coordinate gives the position of the top edge of the layout.

"center" The Y coordinate gives the position of the center of the layout.

"baseline" The Y coordinate gives the position of the baseline of the first line of text in the layout.

"bottom" (default) The Y coordinate gives the position of the bottom edge of the layout.

For the purposes of calculating the position resulting from this alignment, the height of the layout is taken to be the smaller of *height* and *content_height*.

See also *content_valign*.

Type str

HTMLLabel.**background_group** = OrderedGroup(0)

HTMLLabel.**bold**

Bold font style.

Type bool

HTMLLabel.**color**

Text color.

Color is a 4-tuple of RGBA components, each in range [0, 255].

Type (int, int, int, int)

HTMLLabel.**content_valign**

Vertical alignment of content within larger layout box.

This property determines how content is positioned within the layout box when *content_height* is less than *height*. It is one of the enumerants:

top (default) Content is aligned to the top of the layout box.

center Content is centered vertically within the layout box.

bottom Content is aligned to the bottom of the layout box.

This property has no effect when *content_height* is greater than *height* (in which case the content is aligned to the top) or when *height* is None (in which case there is no vertical layout box dimension).

Type str

HTMLLabel.**document**

HTMLLabel.**dpi**

Get DPI used by this layout.

Read-only.

Type float

`HTMLLabel.font_name`

Font family name.

The font name, as passed to `pyglet.font.load`. A list of names can optionally be given: the first matching font will be used.

Type str or list

`HTMLLabel.font_size`

Font size, in points.

Type float

`HTMLLabel.foreground_decoration_group = TextLayoutForegroundDecorationGroup(2)`

`HTMLLabel.foreground_group = TextLayoutForegroundGroup(1)`

`HTMLLabel.height`

Height of the layout.

Type int

`HTMLLabel.italic`

Italic font style.

Type bool

`HTMLLabel.multiline`

Set if multiline layout is enabled.

If multiline is False, newline and paragraph characters are ignored and text is not word-wrapped. If True, the text is word-wrapped only if the `wrap_lines` is True.

Type bool

`HTMLLabel.top_group = <pyglet.text.layout.TextLayoutGroup object>`

`HTMLLabel.width`

Width of the layout.

This property has no effect if `multiline` is False or `wrap_lines` is False.

Type int

`HTMLLabel.x`

X coordinate of the layout.

See also `anchor_x`.

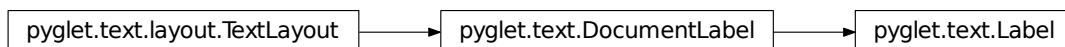
Type int

`HTMLLabel.y`

Y coordinate of the layout.

See also `anchor_y`.

Type int



Label Class

```
class Label (text='', font_name=None, font_size=None, bold=False, italic=False, color=(255, 255, 255, 255), x=0, y=0, width=None, height=None, anchor_x='left', anchor_y='baseline', align='left', multiline=False, dpi=None, batch=None, group=None)
```

Plain text label.

Attributes:

<code>anchor_x</code>	Horizontal anchor alignment.
<code>anchor_y</code>	Vertical anchor alignment.
<code>bold</code>	Bold font style.
<code>color</code>	Text color.
<code>content_valign</code>	Vertical alignment of content within larger layout box.
<code>document</code>	
<code>dpi</code>	Get DPI used by this layout.
<code>font_name</code>	Font family name.
<code>font_size</code>	Font size, in points.
<code>height</code>	Height of the layout.
<code>italic</code>	Italic font style.
<code>multiline</code>	Set if multiline layout is enabled.
<code>text</code>	The text of the label.
<code>width</code>	Width of the layout.
<code>x</code>	X coordinate of the layout.
<code>y</code>	Y coordinate of the layout.

Inherited members**Attributes****Label.anchor_x**

Horizontal anchor alignment.

This property determines the meaning of the x coordinate. It is one of the enumerants:

"left" (default) The X coordinate gives the position of the left edge of the layout.

"center" The X coordinate gives the position of the center of the layout.

"right" The X coordinate gives the position of the right edge of the layout.

For the purposes of calculating the position resulting from this alignment, the width of the layout is taken to be *width* if *multiline* is True and *wrap_lines* is True, otherwise *content_width*.

Type str

Label.anchor_y

Vertical anchor alignment.

This property determines the meaning of the y coordinate. It is one of the enumerants:

"top" The Y coordinate gives the position of the top edge of the layout.

"center" The Y coordinate gives the position of the center of the layout.

"baseline" The Y coordinate gives the position of the baseline of the first line of text in the layout.

"bottom" (default) The Y coordinate gives the position of the bottom edge of the layout.

For the purposes of calculating the position resulting from this alignment, the height of the layout is taken to be the smaller of *height* and *content_height*.

See also *content_valign*.

Type str

`Label.background_group = OrderedGroup(0)`

`Label.bold`

Bold font style.

Type bool

`Label.color`

Text color.

Color is a 4-tuple of RGBA components, each in range [0, 255].

Type (int, int, int, int)

`Label.content_valign`

Vertical alignment of content within larger layout box.

This property determines how content is positioned within the layout box when *content_height* is less than *height*. It is one of the enumerants:

top (default) Content is aligned to the top of the layout box.

center Content is centered vertically within the layout box.

bottom Content is aligned to the bottom of the layout box.

This property has no effect when *content_height* is greater than *height* (in which case the content is aligned to the top) or when *height* is None (in which case there is no vertical layout box dimension).

Type str

`Label.document`

`Label.dpi`

Get DPI used by this layout.

Read-only.

Type float

`Label.font_name`

Font family name.

The font name, as passed to *pyglet.font.load*. A list of names can optionally be given: the first matching font will be used.

Type str or list

`Label.font_size`

Font size, in points.

Type float

`Label.foreground_decoration_group = TextLayoutForegroundDecorationGroup(2)`

`Label.foreground_group = TextLayoutForegroundGroup(1)`

`Label.height`

Height of the layout.

Type int

`Label.italic`

Italic font style.

Type bool

`Label.multiline`

Set if multiline layout is enabled.

If multiline is False, newline and paragraph characters are ignored and text is not word-wrapped. If True, the text is word-wrapped only if the *wrap_lines* is True.

Type bool

`Label.text`

The text of the label.

Type str

`Label.top_group = <pyglet.text.layout.TextLayoutGroup object>`

`Label.width`

Width of the layout.

This property has no effect if *multiline* is False or *wrap_lines* is False.

Type int

`Label.x`

X coordinate of the layout.

See also *anchor_x*.

Type int

`Label.y`

Y coordinate of the layout.

See also *anchor_y*.

Type int

Exceptions

DocumentDecodeException An error occurred decoding document text.

pyglet.text.DocumentDecodeException

DocumentDecodeException

Exception defined in *pyglet.text*

exception DocumentDecodeException

An error occurred decoding document text.

Functions

<code>decode_attributed(text)</code>	Create a document directly from some attributed text.
<code>decode_html(text[, location])</code>	Create a document directly from some HTML formatted text.
<code>decode_text(text)</code>	Create a document directly from some plain text.
<code>get_decoder(filename[, mimetype])</code>	Get a document decoder for the given filename and MIME type.
<code>load(filename[, file, mimetype])</code>	Load a document from a file.

`decode_attributed` Function Defined in `pyglet.text`

`decode_attributed` (*text*)

Create a document directly from some attributed text.

See `pyglet.text.formats.attributed` for a description of attributed text.

Parameters **text** (*str*) – Attributed text to decode.

Return type *FormattedDocument*

`decode_html` Function Defined in `pyglet.text`

`decode_html` (*text*, *location=None*)

Create a document directly from some HTML formatted text.

Parameters

- **text** (*str*) – HTML data to decode.
- **location** (*str*) – Location giving the base path for additional resources referenced from the document (e.g., images).

Return type *FormattedDocument*

`decode_text` Function Defined in `pyglet.text`

`decode_text` (*text*)

Create a document directly from some plain text.

Parameters **text** (*str*) – Plain text to initialise the document with.

Return type *UnformattedDocument*

`get_decoder` Function Defined in `pyglet.text`

`get_decoder` (*filename*, *mimetype=None*)

Get a document decoder for the given filename and MIME type.

If *mimetype* is omitted it is guessed from the filename extension.

The following MIME types are supported:

text/plain Plain text

text/html HTML 4 Transitional

text/vnd.pyglet-attributed Attributed text; see *pyglet.text.formats.attributed*

DocumentDecodeException is raised if another MIME type is given.

Parameters

- **filename** (*str*) – Filename to guess the MIME type from. If a MIME type is given, the filename is ignored.
- **mimetype** (*str*) – MIME type to lookup, or *None* to guess the type from the filename.

Return type *DocumentDecoder*

load Function Defined in *pyglet.text*

load (*filename*, *file=None*, *mimetype=None*)

Load a document from a file.

Parameters

- **filename** (*str*) – Filename of document to load.
- **file** (*file-like object*) – File object containing encoded data. If omitted, *filename* is loaded from disk.
- **mimetype** (*str*) – MIME type of the document. If omitted, the filename extension is used to guess a MIME type. See *get_decoder* for a list of supported MIME types.

Return type *AbstractDocument*

Notes

Defined

- `os`
- `pyglet`

pyglet.window

Windowing and user-interface events.

This module allows applications to create and display windows with an OpenGL context. Windows can be created with a variety of border styles or set fullscreen.

You can register event handlers for keyboard, mouse and window events. For games and kiosks you can also restrict the input to your windows, for example disabling users from switching away from the application with certain key combinations or capturing and hiding the mouse.

Getting started

Call the Window constructor to create a new window:

```
from pyglet.window import Window
win = Window(width=640, height=480)
```

Attach your own event handlers:


```
@win.event
def on_key_press(symbol, modifiers):
    # ... handle this event ...
```

Place drawing code for the window within the *Window.on_draw* event handler:

```
@win.event
def on_draw():
    # ... drawing code ...
```

Call *pyglet.app.run* to enter the main event loop (by default, this returns when all open windows are closed):

```
from pyglet import app
app.run()
```

Creating a game window

Use *Window.set_exclusive_mouse* to hide the mouse cursor and receive relative mouse movement events. Specify *fullscreen=True* as a keyword argument to the *Window* constructor to render to the entire screen rather than opening a window:

```
win = Window(fullscreen=True)
win.set_exclusive_mouse()
```

Working with multiple screens

By default, fullscreen windows are opened on the primary display (typically set by the user in their operating system settings). You can retrieve a list of attached screens and select one manually if you prefer. This is useful for opening a fullscreen window on each screen:

```
display = window.get_platform().get_default_display()
screens = display.get_screens()
windows = []
for screen in screens:
    windows.append(window.Window(fullscreen=True, screen=screen))
```

Specifying a screen has no effect if the window is not fullscreen.

Specifying the OpenGL context properties

Each window has its own context which is created when the window is created. You can specify the properties of the context before it is created by creating a “template” configuration:

```
from pyglet import gl
# Create template config
config = gl.Config()
config.stencil_size = 8
config.aux_buffers = 4
# Create a window using this config
win = window.Window(config=config)
```

To determine if a given configuration is supported, query the screen (see above, “Working with multiple screens”):

```
configs = screen.get_matching_configs(config)
if not configs:
    # ... config is not supported
else:
    win = window.Window(config=configs[0])
```

Modules

<i>event</i>	Events for <i>pyglet.window</i> .
<i>key</i>	Key constants and utilities for <i>pyglet.window</i> .
<i>mouse</i>	Mouse constants and utilities for <i>pyglet.window</i> .

pyglet.window.event Events for *pyglet.window*.

See *Window* for a description of the window event types.

<i>WindowEventLogger</i>	Print all events to a file.
<i>WindowExitHandler</i>	Determine if the window should be closed.

Classes

`pyglet.window.event.WindowEventLogger`

WindowEventLogger Class

class WindowEventLogger (*logfile=None*)

Print all events to a file.

When this event handler is added to a window it prints out all events and their parameters; useful for debugging or discovering which events you need to handle.

Example:

```
win = window.Window()
win.push_handlers(WindowEventLogger())
```

`pyglet.window.event.WindowExitHandler`

WindowExitHandler Class

class WindowExitHandler

Determine if the window should be closed.

This event handler watches for the ESC key or the window close event and sets *self.has_exit* to True when either is pressed. An instance of this class is automatically attached to all new *pyglet.window.Window* objects.

Warning: Deprecated. This class's functionality is provided directly on *Window* in pyglet 1.1.

Variables *has_exit* – True if the user wants to close the window.

Attributes:

has_exit

Attributes

`WindowExitHandler.has_exit = False`

Variables

`print_function = _Feature((2, 6, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 65536)`

Defined

Notes

- key
- mouse
- sys

pyglet.window.key Key constants and utilities for pyglet.window.

Usage:

```
from pyglet.window import Window
from pyglet.window import key

window = Window()

@window.event
def on_key_press(symbol, modifiers):
    # Symbolic names:
    if symbol == key.RETURN:

    # Alphabet keys:
    elif symbol == key.Z:

    # Number keys:
    elif symbol == key._1:

    # Number keypad keys:
    elif symbol == key.NUM_1:

    # Modifiers:
    if modifiers & key.MOD_CTRL:
```

KeyStateHandler Simple handler that tracks the state of keys on the keyboard.

Classes

pyglet.window.key.KeyStateHandler

KeyStateHandler Class

class **KeyStateHandler**

Simple handler that tracks the state of keys on the keyboard. If a key is pressed then this handler holds a True value for it.

For example:

```
>>> win = window.Window
>>> keyboard = key.KeyStateHandler()
>>> win.push_handlers(keyboard)

# Hold down the "up" arrow...

>>> keyboard[key.UP]
True
>>> keyboard[key.DOWN]
False
```

Methods:

<code>clear()</code>	-> None. Remove all items from D.)
<code>copy()</code>	-> a shallow copy of D)
<code>get((k[,d])</code>	-> D[k] if k in D, ...)
<code>items(...)</code>	
<code>keys(...)</code>	
<code>pop((k[,d])</code>	-> v, ...)
<code>popitem()</code>	-> (k, v), ...)
<code>setdefault((k[,d])</code>	-> D.get(k,d), ...)
<code>update([E, ...)</code>	
<code>values(...)</code>	

Inherited members

Methods

`KeyStateHandler.clear()` → None. Remove all items from D.

`KeyStateHandler.copy()` → a shallow copy of D

`KeyStateHandler.get(k[, d])` → `D[k]` if `k` in `D`, else `d`. `d` defaults to `None`.

`KeyStateHandler.items()` → a set-like object providing a view on `D`'s items

`KeyStateHandler.keys()` → a set-like object providing a view on `D`'s keys

`KeyStateHandler.pop(k[, d])` → `v`, remove specified key and return the corresponding value.
If key is not found, `d` is returned if given, otherwise `KeyError` is raised

`KeyStateHandler.popitem()` → (`k`, `v`), remove and return some (key, value) pair as a 2-tuple; but raise `KeyError` if `D` is empty.

`KeyStateHandler.setdefault(k[, d])` → `D.get(k, d)`, also set `D[k]=d` if `k` not in `D`

`KeyStateHandler.update([E], **F)` → `None`. Update `D` from dict/iterable `E` and `F`.
If `E` is present and has a `.keys()` method, then does: for `k` in `E`: `D[k] = E[k]` If `E` is present and lacks a `.keys()` method, then does: for `k`, `v` in `E`: `D[k] = v` In either case, this is followed by: for `k` in `F`: `D[k] = F[k]`

`KeyStateHandler.values()` → an object providing a view on `D`'s values

<code>modifiers_string(modifiers)</code>	Return a string describing a set of modifiers.
<code>motion_string(motion)</code>	Return a string describing a text motion.
<code>symbol_string(symbol)</code>	Return a string describing a key symbol.
<code>user_key(scancode)</code>	Return a key symbol for a key not supported by pyglet.

Functions

`modifiers_string` Function Defined in `pyglet.window.key`

`modifiers_string(modifiers)`

Return a string describing a set of modifiers.

Example:

```
>>> modifiers_string(MOD_SHIFT | MOD_CTRL)
'MOD_SHIFT|MOD_CTRL'
```

Parameters `modifiers` (`int`) – Bitwise combination of modifier constants.

Return type `str`

`motion_string` Function Defined in `pyglet.window.key`

`motion_string(motion)`

Return a string describing a text motion.

Example:

```
>>> motion_string(MOTION_NEXT_WORD)
'MOTION_NEXT_WORD'
```

Parameters `motion` (`int`) – Text motion constant.

Return type `str`

***symbol_string* Function** Defined in *pyglet.window.key*

symbol_string(*symbol*)

Return a string describing a key symbol.

Example:

```
>>> symbol_string(BACKSPACE)
'BACKSPACE'
```

Parameters **symbol** (*int*) – Symbolic key constant.

Return type str

***user_key* Function** Defined in *pyglet.window.key*

user_key(*scancode*)

Return a key symbol for a key not supported by pyglet.

This can be used to map virtual keys or scancodes from unsupported keyboard layouts into a machine-specific symbol. The symbol will be meaningless on any other machine, or under a different keyboard layout.

Applications should use user-keys only when user explicitly binds them (for example, mapping keys to actions in a game options screen).

Variables

compat_platform = 'linux'

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

pyglet.window.mouse Mouse constants and utilities for pyglet.window.

buttons_string(*buttons*) Return a string describing a set of active mouse buttons.

Functions

***buttons_string* Function** Defined in *pyglet.window.mouse*

buttons_string(*buttons*)

Return a string describing a set of active mouse buttons.

Example:

```
>>> buttons_string(LEFT | RIGHT)
'LEFT|RIGHT'
```

Parameters **buttons** (*int*) – Bitwise combination of mouse button constants.

Return type str

Variables

LEFT = 1

`int(x=0) -> integer int(x, base=10) -> integer`

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return `x.__int__()`. For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. `>>> int('0b100', base=0) 4`

MIDDLE = 2

`int(x=0) -> integer int(x, base=10) -> integer`

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return `x.__int__()`. For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. `>>> int('0b100', base=0) 4`

RIGHT = 4

`int(x=0) -> integer int(x, base=10) -> integer`

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return `x.__int__()`. For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. `>>> int('0b100', base=0) 4`

Classes

<i>DefaultMouseCursor</i>	The default mouse cursor used by the operating system.
<i>Display</i>	A display device supporting one or more screens.
<i>FPSDisplay</i>	Display of a window's framerate.
<i>ImageMouseCursor</i>	A user-defined mouse cursor created from an image.
<i>MouseCursor</i>	An abstract mouse cursor.
<i>Platform</i>	Operating-system-level functionality.
<i>Window</i>	Platform-independent application window.

`pyglet.window.MouseCursor`

`pyglet.window.DefaultMouseCursor`

DefaultMouseCursor Class

class `DefaultMouseCursor`

The default mouse cursor used by the operating system.

Attributes:

`drawable`

Attributes`DefaultMouseCursor.drawable = False`

pyglet.window.Display

Display Class**class Display**

A display device supporting one or more screens.

Use `Platform.get_display` or `Platform.get_default_display` to obtain an instance of this class. Use a display to obtain *Screen* instances.

Warning: Deprecated. Use `pyglet.canvas.Display`.

pyglet.window.FPSDisplay

FPSDisplay Class**class FPSDisplay** (*window*)

Display of a window's framerate.

This is a convenience class to aid in profiling and debugging. Typical usage is to create an *FPSDisplay* for each window, and draw the display at the end of the windows' *on_draw* event handler:

```
window = pyglet.window.Window()
fps_display = FPSDisplay(window)

@window.event
def on_draw():
    # ... perform ordinary window drawing operations ...

    fps_display.draw()
```

The style and position of the display can be modified via the *label* attribute. Different text can be substituted by overriding the *set_fps* method. The display can be set to update more or less often by setting the *update_period* attribute.

Variables `label` – The text label displaying the framerate.

Attributes:

`update_period` Time in seconds between updates.

Attributes

`FPSDisplay.update_period = 0.25`

Time in seconds between updates.

Type float

pyglet.window.MouseCursor

pyglet.window.ImageMouseCursor

ImageMouseCursor Class

class `ImageMouseCursor` (*image*, *hot_x=0*, *hot_y=0*)

A user-defined mouse cursor created from an image.

Use this class to create your own mouse cursors and assign them to windows. There are no constraints on the image size or format.

Attributes:

`drawable`

Attributes

`ImageMouseCursor.drawable = True`

pyglet.window.MouseCursor

MouseCursor Class

class `MouseCursor`

An abstract mouse cursor.

Attributes:

drawable Indicates if the cursor is drawn using OpenGL.

Attributes

`MouseCursor.drawable = True`

Indicates if the cursor is drawn using OpenGL. This is True for all mouse cursors except system cursors.

`pyglet.window.Platform`

Platform Class

class Platform

Operating-system-level functionality.

The platform instance can only be obtained with *get_platform*. Use the platform to obtain a *Display* instance.

Warning: Deprecated. Use *pyglet.canvas.Display*

`pyglet.event.EventDispatcher`

`pyglet.window.Window`

Window Class

class Window (*width=None, height=None, caption=None, resizable=False, style=None, fullscreen=False, visible=True, vsync=True, display=None, screen=None, config=None, context=None, mode=None*)

Platform-independent application window.

A window is a “heavyweight” object occupying operating system resources. The “client” or “content” area of a window is filled entirely with an OpenGL viewport. Applications have no access to operating system widgets or controls; all rendering must be done via OpenGL.

Windows may appear as floating regions or can be set to fill an entire screen (fullscreen). When floating, windows may appear borderless or decorated with a platform-specific frame (including, for example, the title bar, minimize and close buttons, resize handles, and so on).

While it is possible to set the location of a window, it is recommended that applications allow the platform to place it according to local conventions. This will ensure it is not obscured by other windows, and appears on an appropriate screen for the user.

To render into a window, you must first call *switch_to*, to make it the current OpenGL context. If you use only one window in the application, there is no need to do this.

Variables `has_exit` – True if the user has attempted to close the window.

Warning: Deprecated. Windows are closed immediately by the default `on_close` handler when `pyglet.app.event_loop` is being used.

Methods:

Attributes:

<code>CURSOR_CROSSHAIR</code>	
<code>CURSOR_DEFAULT</code>	
<code>CURSOR_HAND</code>	
<code>CURSOR_HELP</code>	
<code>CURSOR_NO</code>	
<code>CURSOR_SIZE</code>	
<code>CURSOR_SIZE_DOWN</code>	
<code>CURSOR_SIZE_DOWN_LEFT</code>	
<code>CURSOR_SIZE_DOWN_RIGHT</code>	
<code>CURSOR_SIZE_LEFT</code>	
<code>CURSOR_SIZE_LEFT_RIGHT</code>	
<code>CURSOR_SIZE_RIGHT</code>	
<code>CURSOR_SIZE_UP</code>	
<code>CURSOR_SIZE_UP_DOWN</code>	
<code>CURSOR_SIZE_UP_LEFT</code>	
<code>CURSOR_SIZE_UP_RIGHT</code>	
<code>CURSOR_TEXT</code>	
<code>CURSOR_WAIT</code>	
<code>CURSOR_WAIT_ARROW</code>	
<code>WINDOW_STYLE_BORDERLESS</code>	
<code>WINDOW_STYLE_DEFAULT</code>	
<code>WINDOW_STYLE_DIALOG</code>	
<code>WINDOW_STYLE_TOOL</code>	
<code>caption</code>	The window caption (title).
<code>config</code>	A GL config describing the context of this window.
<code>context</code>	The OpenGL context attached to this window.
<code>display</code>	The display this window belongs to.
<code>event_types</code>	
<code>fullscreen</code>	True if the window is currently fullscreen.
<code>has_exit</code>	
<code>height</code>	The height of the window, in pixels.
<code>invalid</code>	
<code>resizeable</code>	True if the window is resizable.
<code>screen</code>	The screen this window is fullscreen in.
<code>style</code>	The window style; one of the <code>WINDOW_STYLE_*</code> constants.
<code>visible</code>	True if the window is currently visible.
<code>vsync</code>	True if buffer flips are synchronised to the screen's vertical retrace.
<code>width</code>	The width of the window, in pixels.

Attributes

Window.**CURSOR_CROSSHAIR** = 'crosshair'

Window.**CURSOR_DEFAULT** = None

Window.**CURSOR_HAND** = 'hand'

Window.**CURSOR_HELP** = 'help'

Window.**CURSOR_NO** = 'no'

Window.**CURSOR_SIZE** = 'size'

Window.**CURSOR_SIZE_DOWN** = 'size_down'

Window.**CURSOR_SIZE_DOWN_LEFT** = 'size_down_left'

Window.**CURSOR_SIZE_DOWN_RIGHT** = 'size_down_right'

Window.**CURSOR_SIZE_LEFT** = 'size_left'

Window.**CURSOR_SIZE_LEFT_RIGHT** = 'size_left_right'

Window.**CURSOR_SIZE_RIGHT** = 'size_right'

Window.**CURSOR_SIZE_UP** = 'size_up'

Window.**CURSOR_SIZE_UP_DOWN** = 'size_up_down'

Window.**CURSOR_SIZE_UP_LEFT** = 'size_up_left'

Window.**CURSOR_SIZE_UP_RIGHT** = 'size_up_right'

Window.**CURSOR_TEXT** = 'text'

Window.**CURSOR_WAIT** = 'wait'

Window.**CURSOR_WAIT_ARROW** = 'wait_arrow'

Window.**WINDOW_STYLE_BORDERLESS** = 'borderless'

Window.**WINDOW_STYLE_DEFAULT** = None

Window.**WINDOW_STYLE_DIALOG** = 'dialog'

Window.**WINDOW_STYLE_TOOL** = 'tool'

Window.**caption**

The window caption (title). Read-only.

Type str

Window.**config**

A GL config describing the context of this window. Read-only.

Type *pyglet.gl.Config*

Window.**context**

The OpenGL context attached to this window. Read-only.

Type *pyglet.gl.Context*

Window.**display**

The display this window belongs to. Read-only.

Type *Display*

Window.**event_types** = ['on_key_press', 'on_key_release', 'on_text', 'on_text_motion', 'on_text_motion_select', 'on_mouse

`Window.fullscreen`

True if the window is currently fullscreen. Read-only.

Type bool

`Window.has_exit = False`

`Window.height`

The height of the window, in pixels. Read-write.

Type int

`Window.invalid = True`

`Window.resizeable`

True if the window is resizable. Read-only.

Type bool

`Window.screen`

The screen this window is fullscreen in. Read-only.

Type *Screen*

`Window.style`

The window style; one of the `WINDOW_STYLE_*` constants. Read-only.

Type int

`Window.visible`

True if the window is currently visible. Read-only.

Type bool

`Window.vsync`

True if buffer flips are synchronised to the screen's vertical retrace. Read-only.

Type bool

`Window.width`

The width of the window, in pixels. Read-write.

Type int

Inherited members

Methods

`Window.register_event_type(name)`

Register an event type with the dispatcher.

Registering event types allows the dispatcher to validate event handler names as they are attached, and to search attached objects for suitable handlers.

Parameters `name` (*str*) – Name of the event to register.

Exceptions

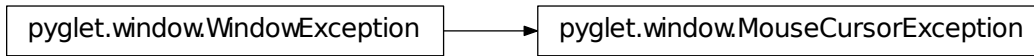
MouseEventException

The root exception for all mouse cursor-related errors.

Continued on next page

Table 2.228 – continued from previous page

<i>NoSuchConfigException</i>	An exception indicating the requested configuration is not available.
<i>NoSuchDisplayException</i>	An exception indicating the requested display is not available.
<i>NoSuchScreenModeException</i>	An exception indicating the requested screen resolution could not be met.
<i>WindowException</i>	The root exception for all window-related errors.



MouseCursorException Exception defined in `pyglet.window`

exception MouseCursorException

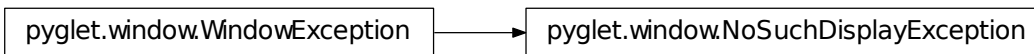
The root exception for all mouse cursor-related errors.



NoSuchConfigException Exception defined in `pyglet.window`

exception NoSuchConfigException

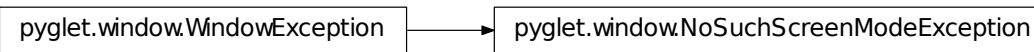
An exception indicating the requested configuration is not available.



NoSuchDisplayException Exception defined in `pyglet.window`

exception NoSuchDisplayException

An exception indicating the requested display is not available.



NoSuchScreenModeException Exception defined in `pyglet.window`

exception NoSuchScreenModeException

An exception indicating the requested screen resolution could not be met.

`pyglet.window.WindowException`

WindowException

Exception defined in `pyglet.window`

exception WindowException

The root exception for all window-related errors.

Functions

`get_platform()` Get an instance of the Platform most appropriate for this system.

get_platform Function Defined in `pyglet.window`

get_platform()

Get an instance of the Platform most appropriate for this system.

Warning: Deprecated. Use `pyglet.canvas.Display`.

Return type *Platform*

Returns The platform instance.

Variables

division = `_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)`

Notes

Defined

- `gl`
- `pyglet`
- `sys`

Development guide

These documents describe details on how to develop pyglet itself further. Read these to get a more detailed insight in how pyglet is designed and how to help making pyglet even better.

Development environment

To develop pyglet, you need an environment with at least the following:

- python 2.7
- python 3.4 and/or 3.5
- py.test
- Your favorite Python editor or IDE

To generate documentation you also need:

- Sphinx
- Graphviz

To use and test all pyglet functionality you should also have:

- AVbin
- Pillow
- coverage

It is preferred to create a virtual Python environment to develop in. It allows you to easily test on all Python version supported by pyglet, not pollute your local system with pyglet development dependencies and not have your local system interfere with pyglet development. This section will show you how to set up a virtual environment for developing pyglet.

Linux or Mac OSX

Setting up

Setting up a virtual environment is almost the same for Linux and OSX. First use your OS's package manager (apt, brew, etc) to install the following dependencies:

- Python 2.7
- Python 3.4 and/or 3.5

- pip (for each version of Python)
- Graphviz (if you want to generate documentation)

[Optional] Make sure pip is the latest version (you might need to add sudo):

```
pip install --upgrade pip
```

Install virtualenv to create virtual environments (you might need to add sudo):

```
pip install virtualenv
```

Make a directory to hold our virtual environments:

```
mkdir venv
```

Now repeat the following steps for each version of Python.

Create a virtual environment (substitute the version number for the version of Python you are creating the environment for:

```
virtualenv -p python2.7 venv/py27
```

Activate the virtual environment:

```
. venv/py27/bin/activate
```

You will see the name of the virtual environment at the start of the command prompt.

Now install required dependencies:

```
pip install pytest
```

Install the documentation generator:

```
pip install sphinx
```

And optional dependencies:

```
pip install pytest-cov Pillow
```

Using

To switch to a virtual environment run the following:

```
. venv/<name of environment>/bin/activate
```

E.g.:

```
. venv/py34/bin/activate
```

To get out of the virtual environment run:

```
deactivate
```

Windows

Setting up

Make sure you download and install:

- Python 2.7
- Python 3.4 and/or 3.5
- Graphviz

Pip should be installed automatically with the latest Python installers. Make sure you do not choose to not install pip.

Now open a command prompt. Repeat the following steps for each version of Python (replace 2.7 with the version you want to use).

[Optional] Make sure pip is the latest version:

```
py -2.7 -m pip install --upgrade pip
```

Install virtualenv to create virtual environments:

```
py -2.7 -m pip install virtualenv
```

Make a directory to hold our virtual environments:

```
md venv
```

Create a virtual environment (substitute the version number for the version of Python you are creating the environment for:

```
py -2.7 -m virtualenv venv\py27
```

Activate the virtual environment:

```
venv\py27\Scripts\activate
```

You will see the name of the virtual environment at the start of the command prompt.

Now install required dependencies:

```
pip install pytest
```

Install the documentation generator:

```
pip install sphinx
```

And optional dependencies:

```
pip install pytest-cov Pillow
```

Using

To switch to a virtual environment run the following:

```
venv<name of environment>\Scripts\activate
```

E.g.:

```
venv\py34\Scripts\activate
```

To get out of the virtual environment run:

```
deactivate
```

Testing pyglet

Test Suites

Tests for pyglet are divided into 3 suites.

Unit tests

Unit tests only cover a single unit of code or a combination of a limited number of units. No resource intensive computations should be included. These tests should run in limited time without any user interaction.

Integration tests

Integration tests cover the integration of components of pyglet into the whole of pyglet and the integration into the supported systems. Like unit tests these tests do not require user interaction, but they can take longer to run due to access to system resources.

Interactive tests

Interactive tests require the user to verify whether the test is successful and in some cases require the user to perform actions in order for the test to continue. These tests can take a long time to run.

There are currently 3 types of interactive test cases:

- tests that can only run in fully interactive mode as they require the user to perform an action in order for the test to continue. These tests are decorated with `requires_user_action()`.
- tests that can run without user interaction, but that cannot validate whether they should pass or fail. These tests are decorated with `requires_user_validation()`.
- tests that can run without user interaction and that can compare results to screenshots from a previous run to determine whether they pass or fail. This is the default type.

Running tests

The pyglet test suite is based on the [py.test framework](#).

It is preferred to use a virtual environment to run the tests. For instructions to set up virtual environments see [Development environment](#). Make sure the virtual environment for the Python version you want to test is active. It is preferred to run the tests on 2.7, 3.4 and 3.5 to make sure changes are compatible with all supported Python versions.

To run all tests, execute `py.test` in the root of the pyglet repository:

```
py.test
```

You can also run just a single suite:

```
py.test tests/unit
py.test tests/integration
py.test tests/interactive
```

For the interactive test suites, there are some extra command line switches for `py.test`:

- `--non-interactive`: Only run the interactive tests that can only verify themselves using screenshots. The screenshots are created when you run the tests in interactive mode, so you will need to run the tests interactively once, before you can use this option;
- `--sanity`: Do a sanity check by running as many interactive tests without user intervention. Not all tests can run without intervention, so these tests will still be skipped. Mostly useful to quickly check changes in code. Not all tests perform complete validation.

Writing tests

Annotations

Some control over test execution can be exerted by using annotations in the form of decorators. One function of annotations is to skip tests under certain conditions.

General annotations

General test annotations are available in the module `tests.annotations`.

@require_platform (*platform*)

Only run the test on the given platform(s), skip on other platforms.

Parameters **platform** (*list(str)*) – A list of platform identifiers as returned by `pyglet.options`. See also `Platform`.

@skip_platform (*platform*)

Skip test on the given platform(s).

Parameters **platform** (*list(str)*) – A list of platform identifiers as returned by `pyglet.options`. See also `Platform`.

@require_gl_extension (*extension*)

Skip the test if the given GL extension is not available.

Parameters **extension** (*str*) – Name of the extension required.

Suite annotations

This is currently not used.

@pytest.mark.unit

Test belongs to the unit test suite.

@pytest.mark.integration

Test belongs to the integration test suite.

@pytest.mark.interactive

Test belongs to the interactive test suite.

Interactive test annotations

Interactive test cases can be marked with specific `py.test` marks. Currently the following marks are used:

@pytest.mark.requires_user_action

Test requires user interaction to run. It needs to be skipped when running in non-interactive or sanity mode.

`@pytest.mark.requires_user_validation`

User validation is required to mark the test passed or failed. However the test can run in sanity mode.

`@pytest.mark.only_interactive`

For another reason the test can only run in interactive mode.

Documentation

This is the `pyglet` documentation, generated with `sphinx`.

Details:

Date	2017/02/21 04:55:17
pyglet version	1.2.2

Note: See the Sphinx warnings log file for errors.

Writing documentation

`pyglet` uses reStructuredText markup language for both the Programming Guide and the docstrings embedded in the code.

Literature

It is divided into several files, which are organized by `toctree` directives .

The entry point for all the documentation is `pyglet/doc/index.txt`, which calls to:

- `pyglet/doc/programming_guide`: The first page of the programming guide.
- `pyglet/doc/internal`: Documentation for those working on Pyglet itself, such as how to make a release or how the `ctypes` library is used.

See also:

- [Sphinx: reStructuredText Primer](#)

Source code

The API documentation is generated from the source code docstrings.

Example

```
class Class1():
    '''Short description.

    Detailed explanation, formatted as reST.
    Can be as detailed as it is needed.

    :Ivariables:
        `arg1`
            description
```

```

:since: pyglet 1.2

'''

attribute1 = None
'''This is an attribute.

More details.
'''

#: This is another attribute.
attribute2 = None


def __init__(self):
    '''Constructor

    :parameters:
        `arg1` : type
            description
    '''

    self.instance_attribute = None
    '''This is an instance attribute.
    '''

def method(self):
    '''Short description.

    :returns: return description
    :rtype: returned type
    '''

def _get_property1(self):
    '''Getter Method

    :return: property1 value
    :rtype: property1 type
    '''

def _set_property1(self, value):
    '''Setter Method

    :param value: property1 value # This is the ReST style
    :type value: property1 type   # But you can use :parameters:
    '''

property1 = property(_get_property1, _set_property1,
                    doc='''Short description

This is another attribute.

:type: type
:see: something else
''')

```

Pyglet has some special roles.

Source	Shows
<code>:deprecated:</code> Do not use	Warning: Deprecated. Do not use
<code>:since:</code> 1.2	Note: Since 1.2

Internal references

To cross-reference to any documented API member, use the following roles:

Source	Reference
<code>:mod: 'pyglet.app'</code>	The <i>pyglet.app</i> module
<code>:func: '~pyglet.app.run'</code>	The <i>run()</i> function
<code>:class: '~pyglet.window.Window'</code>	The <i>Window</i> class
<code>:meth: '~pyglet.window.Window.close'</code>	The <i>close()</i> method
<code>:attr: '~pyglet.window.Window.fullscreen'</code>	The <i>fullscreen</i> attribute

Note: Use ~ to show only the last part.

You can link to arbitrary locations in any document using `:ref:`, but pyglet has a special role `:guide:` for this guide.

A section header of the guide can have an anchor like this:

```
.. _guide_doc_ref:

Internal references
=====
```

It is also possible to put an anchor anywhere:

```
.. _my_anchor:

My anchor.
```

And to insert a reference to an anchor:

Source	Shows
<code>:guide: 'doc_ref'</code>	See also: Programming Guide - <i>Internal references</i>
<code>:ref: 'My Anchor<my_anchor>'</code>	<i>My Anchor</i>

Generation

The complete documentation can be generated using `sphinx`. Make sure you prepare your environment as stated in [Development environment](#).

To build the documentation, execute:

```
./make.py docs
```

Note: Due to a bug in Sphinx, documentation generation currently only works using Python 3.x.

If the build succeeds, the web pages are in `doc/_build/html`

Details

Pyglet documentation system

The documentation build configuration file is `pyglet/doc/conf.py`.

It is a `sphinx` standard configuration file, but adds some requirements of `pyglet` package.

All the modifications to `sphinx` patching are in the `ext` folder.

- Separate Events from regular methods.
- `autosummary` extension: Adds the hidden property and the capability to skip some modules and members

Note: The patching requires a standard `sphinx` version 1.1.3

API Templates

All the `*.rst` files in the `_template` folder configure the layout of the API documentation.

The entry point is `_template/package.rst`.

HTML Theme

The custom `sphinx` theme is in the `ext/theme` folder.

ReST files

The literature is a set of `*.txt` files.

The entry point is `index.txt`.

The `autosummary` directive at `index.txt` directive is mandatory, it generates all the API documentation files.

Omissions

Some things can not be imported when documenting, or are not to be documented,

Skipped members The `skip_member` function in `conf.py` contains rules to prevent certain members to appear in the documentation

Due to the large number of members that were listed when generating, a modification in `autosummary` prevents all members that are not defined in the current module to appear in the member lists.

This means that if a module imports members like this:

```
from pyglet.gl import *
```

That members are not listed in the module documentation.

Warning: There is one exception to the rule, for clarity sake:

- If a member is defined in `module.base`, and imported by `module`, it does appear in the module page lists.

Skipped modules Some modules in `pyglet` can not be imported when documenting, so a black list in `conf.py` contains all the modules that are not to be documented:

- `pyglet.app.carbon`
- `pyglet.app.cocoa`
- `pyglet.app.win32`
- `pyglet.app.xlib`
- `pyglet.canvas.carbon`
- `pyglet.canvas.cocoa`
- `pyglet.canvas.win32`
- `pyglet.canvas.xlib`
- `pyglet.canvas.xlib_vidmoderestore`
- `pyglet.com`
- `pyglet.compat`
- `pyglet.extlibs`
- `pyglet.font.carbon`
- `pyglet.font.fontconfig`
- `pyglet.font.freetype`
- `pyglet.font.freetype_lib`
- `pyglet.font.quartz`
- `pyglet.font.win32`
- `pyglet.font.win32query`
- `pyglet.gl.agl`
- `pyglet.gl.carbon`
- `pyglet.gl.cocoa`
- `pyglet.gl.glext_arb`
- `pyglet.gl.glext_nv`
- `pyglet.gl.glx`
- `pyglet.gl.glx_info`
- `pyglet.gl.glxext_arb`
- `pyglet.gl.glxext_mesa`

- `pyglet.gl.glxext_nv`
- `pyglet.gl.lib_agl`
- `pyglet.gl.lib_glx`
- `pyglet.gl.lib_wgl`
- `pyglet.gl.wgl`
- `pyglet.gl.wgl_info`
- `pyglet.gl.wglext_arb`
- `pyglet.gl.wglext_nv`
- `pyglet.gl.win32`
- `pyglet.gl.xlib`
- `pyglet.image.codecs.gdiplus`
- `pyglet.image.codecs.gdkpixbuf2`
- `pyglet.image.codecs.pil`
- `pyglet.image.codecs.quartz`
- `pyglet.image.codecs.quicktime`
- `pyglet.input.carbon_hid`
- `pyglet.input.carbon_tablet`
- `pyglet.input.darwin_hid`
- `pyglet.input.directinput`
- `pyglet.input.evdev`
- `pyglet.input.wintab`
- `pyglet.input.x11_xinput`
- `pyglet.input.x11_xinput_tablet`
- `pyglet.lib`
- `pyglet.libs`
- `pyglet.media.drivers.directsound`
- `pyglet.media.drivers.openal`
- `pyglet.media.drivers.pulse`
- `pyglet.media.sources.avbin`
- `pyglet.window.carbon`
- `pyglet.window.cocoa`
- `pyglet.window.win32`
- `pyglet.window.xlib`

Note: To be able to document a module, it has to be importable when `sys._is_epydock` is `True`.

Known bugs

- The Window class attributes are not documented because they are defined at BaseWindow class.

Making a pyglet release

1. Clone pyglet into a new directory
2. Make sure it is up to date:

```
hg pull -u
```

3. Update version string in the following files and commit:

- setup.py
- pyglet/__init__.py
- doc/conf.py

4. Tag the current changelist with the version number:

```
hg tag pyglet-x.y.z
```

5. Push the changes to the central repo:

```
hg push
```

6. Build the wheels and documentation:

```
./make.py clean  
./make.py dist
```

7. Upload the wheels, zip and tarball to PyPI:

```
twine upload dist/pyglet-x.y.z*
```

8. Upload the documentation to BitBucket
9. Start a build of the documentation on <https://readthedocs.org/builds/pyglet>
10. Update the download page on BitBucket: <https://bitbucket.org/pyglet/pyglet/wiki/Download>
11. Tell people!

Major version increase

When preparing for a major version you might also want to consider the following:

- Create a maintenance branch for the major version
- Add a readthedocs configuration for that maintenance branch
- Point the url in setup.py to the maintenance branch documentation

OpenGL Interface Implementation

See *OpenGL Interface* for details on the publically-visible modules.

See *ctypes Wrapper Generation* for details on some of these modules are generated.

ctypes linkage

Most functions link to `libGL.so` (Linux), `opengl32.dll` (Windows) or `OpenGL.framework` (OS X). `pyglet.gl.lib` provides some helper types then imports linker functions for the appropriate platform: one of `pyglet.gl.lib_agl`, `pyglet.gl.lib_glx`, `pyglet.gl.lib_wgl`.

On any platform, the following steps are taken to link each function during import:

1. Look in the appropriate library (e.g. `libGL.so`, `libGLU.so`, `opengl32.dll`, etc.) using `cdll` or `windll`.
2. If not found, call `wglGetProcAddress` or `glxGetProcAddress` to try to resolve the function's address dynamically. On OS X, skip this step.
3. On Windows, this will fail if the context hasn't been created yet. Create and return a proxy object `WGLFunctionProxy` which will try the same resolution again when the object is `__call__`'d.

The proxy object caches its result so that subsequent calls have only a single extra function-call overhead.

4. If the function is still not found (either during import or proxy call), the function is replaced with `MissingFunction` (defined in `pyglet.gl.lib`), which raises an exception. The exception message details the name of the function, and optionally the name of the extension it requires and any alternative functions that can be used.

The extension required is currently guessed by `gengl.py` based on nearby `#ifndef` declarations, it is occasionally wrong.

The suggestion list is not currently used, but is intended to be implemented such that calling, for example, `glCreateShader` on an older driver suggests `glCreateShaderObjectARB`, etc.

To access the linking function, import `pyglet.gl.lib` and use one of `link_AGL`, `link_GLX`, `link_WGL`, `link_GL` or `link_GLU`. This is what the generated modules do.

Missing extensions

The latest `glext.h` on opengl.org and [nvidia](http://nvidia.com) does not include some recent extensions listed on the registry. These must be hand coded into `pyglet.gl.glext_missing`. They should be removed when `glext.h` is updated.

ctypes Wrapper Generation

The following modules in `pyglet` are entirely (or mostly) generated from one or more C header files:

- `pyglet.gl.agl`
- `pyglet.gl.gl`
- `pyglet.gl.glext_abi`
- `pyglet.gl.glext_nv`
- `pyglet.gl.glu`
- `pyglet.gl.glx`

- `pyglet.gl.glxext_abi`
- `pyglet.gl.glxext_nv`
- `pyglet.gl.wgl`
- `pyglet.gl.wglext_abi`
- `pyglet.gl.wglext_nv`
- `pyglet.window.xlib.xlib`
- `pyglet.window.xlib.xinerama`

The wrapping framework is in `tools/wraptypes`, and pyglet-specialised batch scripts are `tools/genwrappers.py` (generates xlib wrappers) and `tools/gengl.py` (generates gl wrappers).

Generating GL wrappers

This process needs to be followed when the wraptypes is updated, the header files are updated (e.g., a new release of the operating system), or the GL extensions are updated. Each file can only be generated a a specific platform.

Before beginning, remove the file `tools/.gengl.cache` if it exists. This merely caches header files so they don't need to be repeatedly downloaded (but you'd prefer to use the most recent uncached copies if you're reading this, presumably).

On Linux, generate `pyglet.gl.gl`, `pyglet.gl.glext_abi`, `pyglet.gl.glext_nv` and `pyglet.gl.glu` (the complete user-visible GL package):

```
python tools/gengl.py gl glext_abi glext_nv glu
```

The header files for `pyglet.gl.gl` and `pyglet.gl.glu` are located in `/usr/include/GL`. Ensure your Linux distribution has recent versions of these files (unfortunately they do not seem to be accessible outside of a distribution or OS).

The header files for `pyglet.glext_abi` and `pyglet.glext_nv` are downloaded from <http://www.opengl.org> and <http://developer.nvidia.com>, respectively.

On Linux still, generate `pyglet.gl.glx`, `pyglet.gl.glxext_abi` and `pyglet.gl.glxext_nv`:

```
python tools/gengl.py glx glxext_abi glxext_nv
```

The header file for `pyglet.gl.glx` is in `/usr/include/GL`, and is expected to depend on X11 header files from `/usr/include/X11`. `glext_abi` and `glext_nv` header files are downloaded from the above websites.

On OS X, generate `pyglet.gl.agl`:

```
python tools/gengl.py agl
```

Watch a movie while you wait – it uses virtually every header file on the system. Expect to see one syntax error in `PictUtils.h` line 67, it is unimportant.

On Windows XP, generate `pyglet.gl.wgl`, `pyglet.gl.wglext_abi` and `pyglet.gl.wglext_nv`:

```
python tools/gengl.py wgl wglext_abi wglext_nv
```

You do not need to have a development environment installed on Windows. `pyglet.gl.wgl` is generated from `tools/wgl.h`, which is a hand-coded header file containing the prototypes and constants for WGL and its dependencies. In a real development environment you would find these mostly in `WinGDI.h`, but wraptypes is not quite sophisticated enough to parse Windows system headers (see below for what needs implementing). It is extremely unlikely this header will ever need to change (excepting a bug fix).

The headers for `pyglet.gl.wglext_abi` and `pyglet.gl.wglext_nv` are downloaded from the same websites as for GL and GLX.

Generated GL wrappers

Each generated file contains a pair of markers `# BEGIN GENERATED CONTENT` and `# END GENERATED CONTENT` which are searched for when replacing the file. If either marker is missing or corrupt, the file will not be modified. This allows for custom content around the generated content. Only `glx.py` makes use of this, to include some additional enumerators that are not generated by default.

If a generating process is interrupted (either you get sick of it, or it crashes), it will leave a partially-complete file written, which will not include both markers. It is up to you to restore the file or otherwise reinsert the markers.

Generating Xlib wrappers

On Linux with the Xinerama extension installed (doesn't have to be in use, just available), run:

```
python tools/genwrappers.py
```

This generates `pyglet.window.xlib.xlib` and `pyglet.window.xlib.xinerama`.

Note that this process, as well as the generated modules, depend on `pyglet.gl.glx`. So, you should always run this *after* the above GL generation.

wraptypes

`wraptypes` is a general utility for creating ctypes wrappers from C header files. The front-end is `tools/wraptypes/wrap.py`, for usage:

```
python tools/wraptypes/wrap.py -h
```

There are three components to `wraptypes`:

preprocessor.py Interprets preprocessor declarations and converts the source header files into a list of tokens.

cparser.py Parses the preprocessed tokens for type and function declarations and calls `handle_` methods on the class `CParser` in a similar manner to a SAX parser.

ctypesparser.py Interprets C declarations and types from `CParser` and creates corresponding ctypes declarations, calling `handle_` methods on the class `CtypesParser`.

The front-end `wrap.py` provides a simple subclass of `CtypesParser`, `CtypesWrapper`, which writes the ctypes declarations found to a file in a format that can be imported as a module.

Parser Modifications

The parsers are built upon a modified version of [PLY](#), a Python implementation of `lex` and `yacc`. The modified source is included in the `wraptypes` directory. The modifications are:

- Grammar is abstracted out of `Parser`, so multiple grammars can easily be defined in the same module.
- Tokens and symbols keep track of their filename as well as line number.
- Lexer state can be pushed onto a stack.

The first time the parsers are run (or after they are modified), PLY creates `pptab.py` and `parsetab.py` in the current directory. These are the generated state machines, which can take a few seconds to generate. The file `parser.out` is created if debugging is enabled, and contains the parser description (of the last parser that was generated), which is essential for debugging.

Preprocessor

The grammar and parser are defined in `preprocessor.py`.

There is only one lexer state. Each token has a type which is a string (e.g. `'CHARACTER_CONSTANT'`) and a value. Token values, when read directly from the source file are only ever strings. When tokens are written to the output list they sometimes have tuple values (for example, a `PP_DEFINE` token on output).

Two lexer classes are defined: `PreprocessorLexer`, which reads a stack of files (actually strings) as input, and `TokenListLexer`, which reads from a list of already-parsed tokens (used for parsing expressions).

The preprocessing entry-point is the `PreprocessorParser` class. This creates a `PreprocessorLexer` and its grammar during construction. The system include path includes the GCC search path by default but can be modified by altering the `include_path` and `framework_path` lists. The `system_headers` dict allows header files to be implied on the search path that don't exist. For example, by setting:

```
system_headers['stdlib.h'] = '''#ifndef STDLIB_H
#define STDLIB_H

/* ... */
#endif
'''
```

you can insert your own custom header in place of the one on the filesystem. This is useful when parsing headers from network locations.

Parsing begins when `parse` is called. Specify one or both of a filename and a string of data. If `debug` kwarg is `True`, syntax errors dump the parser state instead of just the line number where they occurred.

The production rules specify the actions; these are implemented in `PreprocessorGrammar`. The actions call methods on `PreprocessorParser`, such as:

- `include(self, header)`, to push another file onto the lexer.
- `include_system(self, header)`, to search the system path for a file to push onto the lexer
- `error(self, message, filename, line)`, to signal a parse error. Not all syntax errors get this far, due to limitations in the parser. A parse error at EOF will just print to `stderr`.
- `write(self, tokens)`, to write tokens to the output list. This is the default action when no preprocessing declaratives are being parsed.

The parser has a stack of `ExecutionState`, which specifies whether the current tokens being parsed are ignored or not (tokens are ignored in an `#if` that evaluates to 0). This is a little more complicated than just a boolean flag: the parser must also ignore `#elif` conditions that can have no effect. The `enable_declaratives` and `enable_elif_conditionals` return `True` if the top-most `ExecutionState` allows declaratives and `#elif` conditionals to be parsed, respectively. The execution state stack is modified with the `condition_*` methods.

`PreprocessorParser` has a `PreprocessorNamespace` which keeps track of the currently defined macros. You can create and specify your own namespace, or use one that is created by default. The default namespace includes GCC platform macros needed for parsing system headers, and some of the STDC macros.

Macros are expanded when tokens are written to the output list, and when conditional expressions are parsed. `PreprocessorNamespace.apply_macros(tokens)` takes care of this, replacing function parameters, vari-

able arguments, macro objects and (mostly) avoiding infinite recursion. It does not yet handle the # and ## operators, which are needed to parse the Windows system headers.

The process for evaluating a conditional (#if or #elif) is:

1. Tokens between PP_IF or PP_ELIF and NEWLINE are expanded by `apply_macros`.
2. The resulting list of tokens is used to construct a `TokenListLexer`.
3. This lexer is used as input to a `ConstantExpressionParser`. This parser uses the `ConstantExpressionGrammar`, which builds up an AST of `ExpressionNode` objects.
4. `parse` is called on the `ConstantExpressionParser`, which returns the resulting top-level `ExpressionNode`, or `None` if there was a syntax error.
5. The `evaluate` method of the `ExpressionNode` is called with the preprocessor's namespace as the evaluation context. This allows the expression nodes to resolve defined operators.
6. The result of `evaluate` is always an int; non-zero values are treated as `True`.

Because pyglet requires special knowledge of the preprocessor declaratives that were encountered in the source, these are encoded as pseudo-tokens within the output token list. For example, after a `#ifndef` is evaluated, it is written to the token list as a `PP_IFNDEF` token.

`#define` is handled specially. After applying it to the namespace, it is parsed as an expression immediately. This is allowed (and often expected) to fail. If it does not fail, a `PP_DEFINE_CONSTANT` token is created, and the value is the result of evaluating the expression. Otherwise, a `PP_DEFINE` token is created, and the value is the string concatenation of the tokens defined. Special handling of parseable expressions makes it simple to later parse constants defined as, for example:

```
#define RED_SHIFT 8
#define RED_MASK (0x0f << RED_SHIFT)
```

The preprocessor can be tested/debugged by running `preprocessor.py` stand-alone with a header file as the sole argument. The resulting token list will be written to `stdout`.

CParser

The lexer for `CParser`, `CLexer`, takes as input a list of tokens output from the preprocessor. The special preprocessor tokens such as `PP_DEFINE` are intercepted here and handled immediately; hence they can appear anywhere in the source header file without causing problems with the parser. At this point `IDENTIFIER` tokens which are found to be the name of a defined type (the set of defined types is updated continuously during parsing) are converted to `TYPE_NAME` tokens.

The entry-point to parsing C source is the `CParser` class. This creates a preprocessor in its constructor, and defines some default types such as `wchar_t` and `__int64_t`. These can be disabled with `kwargs`.

Preprocessing can be quite time-consuming, especially on OS X where thousands of `#include` declaratives are processed when Carbon is parsed. To minimise the time required to parse similar (or the same, while debugging) header files, the token list from preprocessing is cached and reused where possible.

This is handled by `CPreprocessorParser`, which overrides `push_file` to check with `CParser` if the desired file is cached. The cache is checked against the file's modification timestamp as well as a "memento" that describes the currently defined tokens. This is intended to avoid using a cached file that would otherwise be parsed differently due to the defined macros. It is by no means perfect; for example, it won't pick up on a macro that has been defined differently. It seems to work well enough for the header files pyglet requires.

The header cache is saved and loaded automatically in the working directory as `.header.cache`. The cache should be deleted if you make changes to the preprocessor, or are experiencing cache errors (these are usually accompanied by a "what-the?" exclamation from the user).

The actions in the grammar construct parts of a “C object model” and call methods on `CParser`. The C object model is not at all complete, containing only what pyglet (and any other ctypes-wrapping application) requires. The classes in the object model are:

Declaration A single declaration occurring outside of a function body. This includes type declarations, function declarations and variable declarations. The attributes are `declarator` (see below), `type` (a `Type` object) and `storage` (for example, `'typedef'`, `'const'`, `'static'`, `'extern'`, etc).

Declarator A declarator is a thing being declared. Declarators have an `identifier` (the name of it, `None` if the declarator is abstract, as in some function parameter declarations), an optional `initializer` (currently ignored), an optional linked-list of `array` (giving the dimensions of the array) and an optional list of `parameters` (if the declarator is a function).

Pointer This is a type of declarator that is dereferenced via `pointer` to another declarator.

Array Array has size (an int, its dimension, or `None` if unsized) and a pointer `array` to the next array dimension, if any.

Parameter A function parameter consisting of a `type` (`Type` object), `storage` and `declarator`.

Type Type has a list of `qualifiers` (e.g. `'const'`, `'volatile'`, etc) and `specifiers` (the meaty bit).

TypeSpecifier A base `TypeSpecifier` is just a string, such as `'int'` or `'Foo'` or `'unsigned'`. Note that types can have multiple `TypeSpecifiers`; not all combinations are valid.

StructTypeSpecifier This is the specifier for a struct or union (if `is_union` is `True`) type. `tag` gives the optional `foo` in `struct foo` and `declarations` is the meat (an empty list for an opaque or unspecified struct).

EnumSpecifier This is the specifier for an enum type. `tag` gives the optional `foo` in `enum foo` and `enumerators` is the list of `Enumerator` objects (an empty list for an unspecified enum).

Enumerator Enumerators exist only within `EnumSpecifier`. Contains `name` and `expression`, an `ExpressionNode` object.

The `ExpressionNode` object hierarchy is similar to that used in the preprocessor, but more fully-featured, and using a different `EvaluationContext` which can evaluate identifiers and the `sizeof` operator (currently it actually just returns 0 for both).

Methods are called on `CParser` as declarations and preprocessor declaratives are parsed. The are mostly self explanatory. For example:

handle_ifndef(self, name, filename, lineno) An `#ifndef` was encountered testing the macro `name` in file `filename` at line `lineno`.

handle_declaration(self, declaration, filename, lineno) `declaration` is an instance of `Declaration`.

These methods should be overridden by a subclass to provide functionality. The `DebugCParser` does this and prints out the arguments to each `handle_` method.

The `CParser` can be tested in isolation by running it stand-alone with the filename of a header as the sole argument. A `DebugCParser` will be constructed and used to parse the header.

CtypesParser

`CtypesParser` is implemented in `ctypesparser.py`. It is a subclass of `CParser` and implements the `handle_` methods to provide a more ctypes-friendly interpretation of the declarations.

To use, subclass and override the methods:

handle_ctypes_constant(self, name, value, filename, lineno) An integer or float constant (in a `#define`).

handle_ctypes_type_definition(self, name, ctype, filename, lineno) A `typedef` declaration. See below for type of `ctype`.

handle_ctypes_function(self, name, restype, argtypes, filename, lineno) A function declaration with the given return type and argument list.

handle_ctypes_variable(self, name, ctype, filename, lineno) Any other non-`static` declaration.

Types are represented by instances of `CtypesType`. This is more easily manipulated than a “real” `ctypes` type. There are subclasses for `CtypesPointer`, `CtypesArray`, `CtypesFunction`, and so on; see the module for details.

Each `CtypesType` class implements the `visit` method, which can be used, Visitor pattern style, to traverse the type hierarchy. Call the `visit` method of any type with an implementation of `CtypesTypeVisitor`: all pointers, array bases, function parameters and return types are traversed automatically (struct members are not, however).

This is useful when writing the contents of a struct or enum. Before writing a type declaration for a struct type (which would consist only of the struct’s tag), `visit` the type and handle the `visit_struct` method on the visitor to print out the struct’s members first. Similarly for enums.

`ctypesparser.py` can not be run stand-alone. `wrap.py` provides a straight-forward implementation that writes a module of `ctypes` wrappers. It can filter the output based on the originating filename. See the module docstring for usage and extension details.

Related Documentation

- [OpenGL Programming Guide](#)
- [OpenGL Reference Pages](#)
- [AVbin Documentation](#)
- [ctypes Reference](#)
- [Python Documentation](#)

a

`pyglet.app`, 89
`pyglet.app.base`, 90

c

`pyglet.canvas`, 96
`pyglet.canvas.base`, 96
`pyglet.clock`, 102

d

`pyglet.debug`, 107

e

`pyglet.event`, 108

f

`pyglet.font`, 111
`pyglet.font.base`, 112
`pyglet.font.text`, 120
`pyglet.font.ttf`, 124

g

`pyglet.gl`, 127
`pyglet.gl.base`, 127
`pyglet.gl.gl`, 131
`pyglet.gl.gl_info`, 131
`pyglet.gl.glu`, 133
`pyglet.gl.glu_info`, 134
`pyglet.gl.lib`, 135
`pyglet.graphics`, 139
`pyglet.graphics.allocation`, 141
`pyglet.graphics.vertexattribute`, 142
`pyglet.graphics.vertexbuffer`, 148
`pyglet.graphics.vertexdomain`, 154

i

`pyglet.image`, 161
`pyglet.image.atlas`, 163
`pyglet.image.codecs`, 165
`pyglet.image.codecs.bmp`, 165

`pyglet.image.codecs.dds`, 165
`pyglet.image.codecs.gif`, 165
`pyglet.image.codecs.png`, 166
`pyglet.image.codecs.s3tc`, 166
`pyglet.info`, 203
`pyglet.input`, 205
`pyglet.input.base`, 205
`pyglet.input.evdev_constants`, 216

m

`pyglet.media`, 218
`pyglet.media.drivers`, 219
`pyglet.media.drivers.base`, 219
`pyglet.media.drivers.silent`, 219
`pyglet.media.events`, 222
`pyglet.media.exceptions`, 222
`pyglet.media.listener`, 223
`pyglet.media.player`, 224
`pyglet.media.sources`, 226
`pyglet.media.sources.base`, 227
`pyglet.media.sources.loader`, 233
`pyglet.media.sources.procedural`, 235
`pyglet.media.sources.riff`, 244
`pyglet.media.threads`, 249

p

`pyglet`, 79

r

`pyglet.resource`, 251

s

`pyglet.sprite`, 257

t

`pyglet.text`, 262
`pyglet.text.caret`, 263
`pyglet.text.document`, 265
`pyglet.text.formats`, 270
`pyglet.text.formats.attributed`, 271
`pyglet.text.formats.html`, 271

`pyglet.text.formats.plaintext`, [272](#)
`pyglet.text.formats.structured`, [273](#)
`pyglet.text.layout`, [275](#)
`pyglet.text.runlist`, [286](#)

W

`pyglet.window`, [300](#)
`pyglet.window.event`, [302](#)
`pyglet.window.key`, [303](#)
`pyglet.window.mouse`, [306](#)

A

- `absolute_import` (in module `pyglet.font`), 125
- `absolute_import` (in module `pyglet.font.base`), 119
- `absolute_import` (in module `pyglet.font.text`), 123
- `absolute_import` (in module `pyglet.gl`), 138
- `absolute_import` (in module `pyglet.graphics`), 161
- `absolute_import` (in module `pyglet.graphics.vertexattribute`), 148
- `absolute_import` (in module `pyglet.graphics.vertexbuffer`), 153
- `absolute_import` (in module `pyglet.graphics.vertexdomain`), 157
- `absolute_import` (in module `pyglet.image`), 202
- `absolute_import` (in module `pyglet.input`), 218
- `absolute_import` (in module `pyglet.media.drivers`), 222
- `absolute_import` (in module `pyglet.sprite`), 262
- `absolute_import` (in module `pyglet.text.layout`), 286
- `AbsoluteAxis` (class in `pyglet.input.base`), 206
- `AbstractAttribute` (class in `pyglet.graphics.vertexattribute`), 143
- `AbstractAudioDriver` (class in `pyglet.media.drivers.base`), 219
- `AbstractAudioPlayer` (class in `pyglet.media.drivers.base`), 219
- `AbstractBuffer` (class in `pyglet.graphics.vertexbuffer`), 148
- `AbstractBufferRegion` (class in `pyglet.graphics.vertexbuffer`), 149
- `AbstractDocument` (class in `pyglet.text.document`), 267
- `AbstractImage` (class in `pyglet.image`), 169
- `AbstractImageSequence` (class in `pyglet.image`), 170
- `AbstractListener` (class in `pyglet.media.listener`), 224
- `AbstractMappable` (class in `pyglet.graphics.vertexbuffer`), 149
- `AbstractRunIterator` (class in `pyglet.text.runlist`), 287
- `AbstractSourceLoader` (class in `pyglet.media.sources.loader`), 234
- `add_decoders()` (in module `pyglet.image.codecs`), 167
- `add_default_image_codecs()` (in module `pyglet.image.codecs`), 167
- `add_directory()` (in module `pyglet.font`), 125
- `add_encoders()` (in module `pyglet.image.codecs`), 167
- `add_file()` (in module `pyglet.font`), 125
- `add_font` (in module `pyglet.resource`), 255
- `add_font_data()` (`pyglet.font.base.Font` class method), 113
- `ADSREnvelope` (class in `pyglet.media.sources.procedural`), 235
- `advance` (`Glyph` attribute), 114
- `album` (`SourceInfo` attribute), 230
- `Allocator` (class in `pyglet.graphics.allocation`), 141
- `Allocator` (class in `pyglet.image.atlas`), 163
- `AllocatorException`, 164
- `AllocatorMemoryException`, 142
- `anchor_x` (`AbstractImage` attribute), 169
- `anchor_x` (`ScrollableTextLayout` attribute), 280
- `anchor_x` (`TextLayout` attribute), 283
- `anchor_y` (`AbstractImage` attribute), 169
- `anchor_y` (`ScrollableTextLayout` attribute), 280
- `anchor_y` (`TextLayout` attribute), 284
- `Animation` (class in `pyglet.image`), 170
- `animation` (in module `pyglet.resource`), 255
- `AnimationFrame` (class in `pyglet.image`), 171
- `app` (in module `pyglet.app.base`), 92
- `AppException`, 94
- `AppleRemote` (class in `pyglet.input.base`), 207
- `ascent` (`Font` attribute), 113
- `attributed` (in module `pyglet.resource`), 255
- `AttributedTextDecoder` (class in `pyglet.text.formats.attributed`), 271
- `audio_format` (`Source` attribute), 228
- `AudioData` (class in `pyglet.media.sources.base`), 227
- `AudioFormat` (class in `pyglet.media.sources.base`), 228
- `author` (`SourceInfo` attribute), 230
- `AVbinSourceLoader` (class in `pyglet.media.sources.loader`), 234

B

- `background_group` (`TextLayout` attribute), 284
- `BASELINE` (`Text` attribute), 122
- `Batch` (class in `pyglet.graphics`), 158
- `batch` (`Sprite` attribute), 259

bold (DocumentLabel attribute), 290
 BOTTOM (Text attribute), 122
 BufferImage (class in pyglet.image), 171
 BufferImageMask (class in pyglet.image), 172
 BufferManager (class in pyglet.image), 174
 Button (class in pyglet.input.base), 209
 buttons_string() (in module pyglet.window.mouse), 306

C

c_void (class in pyglet.gl.lib), 136
 CannotSeekException, 223
 Canvas (class in pyglet.canvas.base), 96
 CanvasConfig (class in pyglet.gl.base), 128
 caption (Window attribute), 312
 Caret (class in pyglet.text.caret), 263
 CENTER (Text attribute), 122
 CheckerImagePattern (class in pyglet.image), 174
 Clock (class in pyglet.clock), 103
 clock (in module pyglet.app.base), 92
 ClockDisplay (class in pyglet.clock), 104
 color (Caret attribute), 264
 color (DocumentLabel attribute), 290
 color (Sprite attribute), 259
 color (Text attribute), 122
 color_as_bytes() (in module pyglet.image), 201
 ColorAttribute (class in pyglet.graphics.vertexattribute), 144
 ColorBufferImage (class in pyglet.image), 174
 colors (VertexList attribute), 156
 columns (TextureGrid attribute), 192
 comment (SourceInfo attribute), 230
 compat_platform (in module pyglet.app), 95
 compat_platform (in module pyglet.app.base), 92
 compat_platform (in module pyglet.clock), 107
 compat_platform (in module pyglet.font.base), 120
 compat_platform (in module pyglet.font.text), 123
 compat_platform (in module pyglet.gl), 138
 compat_platform (in module pyglet.gl.base), 130
 compat_platform (in module pyglet.graphics), 161
 compat_platform (in module pyglet.graphics.vertexattribute), 148
 compat_platform (in module pyglet.graphics.vertexbuffer), 153
 compat_platform (in module pyglet.graphics.vertexdomain), 157
 compat_platform (in module pyglet.image), 202
 compat_platform (in module pyglet.image.codecs), 168
 compat_platform (in module pyglet.sprite), 262
 compat_platform (in module pyglet.text.layout), 286
 compat_platform (in module pyglet.window.key), 306
 CompressedImageData (class in pyglet.image), 175
 cone_inner_angle (Player attribute), 225
 cone_orientation (Player attribute), 225
 cone_outer_angle (Player attribute), 225

cone_outer_gain (Player attribute), 225
 Config (class in pyglet.gl.base), 129
 config (Window attribute), 312
 ConfigException, 138
 ConstRunIterator (class in pyglet.text.runlist), 288
 content_valign (TextLayout attribute), 284
 Context (class in pyglet.gl.base), 130
 context (Window attribute), 312
 CONTEXT_SHARE_EXISTING (Context attribute), 130
 CONTEXT_SHARE_NONE (Context attribute), 130
 ContextException, 138
 Control (class in pyglet.input.base), 209
 copyright (SourceInfo attribute), 230
 create() (in module pyglet.image), 201
 create() (pyglet.image.Texture class method), 186
 create_attribute() (in module pyglet.graphics.vertexattribute), 147
 create_attribute_usage() (in module pyglet.graphics.vertexdomain), 157
 create_audio_driver() (in module pyglet.media.drivers.silent), 221
 create_buffer() (in module pyglet.graphics.vertexbuffer), 152
 create_domain() (in module pyglet.graphics.vertexdomain), 157
 create_for_image() (pyglet.image.TileableTexture class method), 198
 create_for_image_grid() (pyglet.image.Texture3D class method), 188
 create_for_images() (pyglet.image.Texture3D class method), 188
 create_for_size() (pyglet.image.Texture class method), 186
 create_indexed_domain() (in module pyglet.graphics.vertexdomain), 157
 create_mappable_buffer() (in module pyglet.graphics.vertexbuffer), 153
 CURSOR_CROSSHAIR (Window attribute), 312
 CURSOR_DEFAULT (Window attribute), 312
 CURSOR_HAND (Window attribute), 312
 CURSOR_HELP (Window attribute), 312
 CURSOR_NO (Window attribute), 312
 CURSOR_SIZE (Window attribute), 312
 CURSOR_SIZE_DOWN (Window attribute), 312
 CURSOR_SIZE_DOWN_LEFT (Window attribute), 312
 CURSOR_SIZE_DOWN_RIGHT (Window attribute), 312
 CURSOR_SIZE_LEFT (Window attribute), 312
 CURSOR_SIZE_LEFT_RIGHT (Window attribute), 312
 CURSOR_SIZE_RIGHT (Window attribute), 312
 CURSOR_SIZE_UP (Window attribute), 312
 CURSOR_SIZE_UP_DOWN (Window attribute), 312
 CURSOR_SIZE_UP_LEFT (Window attribute), 312

CURSOR_SIZE_UP_RIGHT (Window attribute), 312
 CURSOR_TEXT (Window attribute), 312
 CURSOR_WAIT (Window attribute), 312
 CURSOR_WAIT_ARROW (Window attribute), 312

D

data (ImageData attribute), 181
 data (ImageDataRegion attribute), 182
 debug (Config attribute), 130
 debug_print() (in module pyglet.debug), 107
 decode_attributed() (in module pyglet.text), 299
 decode_html() (in module pyglet.text), 299
 decode_text() (in module pyglet.text), 299
 decorate_function() (in module pyglet.gl.lib), 137
 DEFAULT_MODE (in module pyglet.gl.gl), 131
 default_style (HTMLDecoder attribute), 272
 DefaultMouseCursor (class in pyglet.window), 307
 depth (ScreenMode attribute), 99
 DepthBufferImage (class in pyglet.image), 176
 DepthTexture (class in pyglet.image), 178
 descent (Font attribute), 113
 Device (class in pyglet.input.base), 211
 DeviceException, 216
 DeviceExclusiveException, 216
 DeviceOpenException, 216
 Digitar (class in pyglet.media.sources.procedural), 236
 display (Canvas attribute), 97
 Display (class in pyglet.canvas.base), 97
 Display (class in pyglet.window), 308
 display (Screen attribute), 98
 display (Window attribute), 312
 displays (in module pyglet.app), 95
 division (in module pyglet.app.base), 92
 division (in module pyglet.clock), 107
 division (in module pyglet.font), 125
 division (in module pyglet.font.text), 123
 division (in module pyglet.font.ttf), 124
 division (in module pyglet.graphics.allocation), 142
 division (in module pyglet.image), 202
 division (in module pyglet.image.atlas), 165
 division (in module pyglet.input.base), 216
 division (in module pyglet.media.drivers.silent), 221
 division (in module pyglet.media.player), 226
 division (in module pyglet.media.sources.base), 233
 division (in module pyglet.media.sources.procedural), 244
 division (in module pyglet.media.sources.riff), 249
 division (in module pyglet.text.formats.structured), 275
 division (in module pyglet.text.layout), 286
 division (in module pyglet.window), 315
 document (TextLayout attribute), 284
 DocumentDecodeException, 298
 DocumentDecoder (class in pyglet.text), 289
 DocumentLabel (class in pyglet.text), 290

dpi (TextLayout attribute), 284
 draw() (in module pyglet.graphics), 160
 draw_indexed() (in module pyglet.graphics), 160
 drawable (DefaultMouseCursor attribute), 308
 drawable (ImageMouseCursor attribute), 309
 drawable (MouseCursor attribute), 310
 dummy (c_void attribute), 136
 dump() (in module pyglet.info), 203
 dump_al() (in module pyglet.info), 203
 dump_avbin() (in module pyglet.info), 204
 dump_gl() (in module pyglet.info), 204
 dump_glu() (in module pyglet.info), 204
 dump_glx() (in module pyglet.info), 204
 dump_media() (in module pyglet.info), 204
 dump_pyglet() (in module pyglet.info), 204
 dump_python() (in module pyglet.info), 204
 dump_window() (in module pyglet.info), 204
 dump_wintab() (in module pyglet.info), 204
 duration (Source attribute), 229

E

edge_flags (VertexList attribute), 156
 EdgeFlagAttribute (class in pyglet.graphics.vertexattribute), 144
 Envelope (class in pyglet.media.sources.procedural), 237
 envelope (ProceduralSource attribute), 239
 errcheck() (in module pyglet.gl.lib), 137
 errcheck_glbeg() (in module pyglet.gl.lib), 137
 errcheck_glend() (in module pyglet.gl.lib), 137
 event (in module pyglet.app.base), 92
 EVENT_HANDLED (in module pyglet.event), 111
 event_loop (in module pyglet.app), 95
 event_types (AbstractDocument attribute), 267
 event_types (AppleRemote attribute), 208
 event_types (Control attribute), 210
 event_types (EventLoop attribute), 91
 event_types (IncrementalTextLayout attribute), 278
 event_types (Joystick attribute), 212
 event_types (Player attribute), 225
 event_types (Sprite attribute), 260
 event_types (TabletCanvas attribute), 215
 event_types (Window attribute), 312
 EVENT_UNHANDLED (in module pyglet.event), 111
 EventBuffer (class in pyglet.media.drivers.silent), 220
 EventDispatcher (class in pyglet.event), 110
 EventException, 111
 EventLoop (class in pyglet.app.base), 90
 exit() (in module pyglet.app), 94
 extensions (GLInfo attribute), 132
 extensions (GLUInfo attribute), 134

F

file (in module pyglet.resource), 255
 FileLocation (class in pyglet.resource), 252

- FilteredRunIterator (class in pyglet.text.runlist), 288
 - FlatEnvelope (class in pyglet.media.sources.procedural), 238
 - FM (class in pyglet.media.sources.procedural), 237
 - fog_coords (VertexList attribute), 156
 - FogCoordAttribute (class in pyglet.graphics.vertexattribute), 144
 - Font (class in pyglet.font.base), 112
 - font (Text attribute), 122
 - font_name (DocumentLabel attribute), 290
 - font_size (DocumentLabel attribute), 290
 - font_sizes (HTMLDecoder attribute), 272
 - FontException, 119
 - foreground_decoration_group (TextLayout attribute), 284
 - foreground_group (TextLayout attribute), 284
 - format (BufferImage attribute), 171
 - format (BufferImageMask attribute), 173
 - format (ColorBufferImage attribute), 174
 - format (DepthBufferImage attribute), 177
 - format (ImageData attribute), 181
 - format_re (OrderedListBuilder attribute), 274
 - FormattedDocument (class in pyglet.text.document), 268
 - forward_compatible (Config attribute), 130
 - forward_orientation (AbstractListener attribute), 224
 - FPSDisplay (class in pyglet.window), 308
 - from_image_sequence() (pyglet.image.Animation class method), 171
 - fullscreen (Window attribute), 312
- ## G
- GenericAttribute (class in pyglet.graphics.vertexattribute), 145
 - genre (SourceInfo attribute), 230
 - get_animation_decoders() (in module pyglet.image.codecs), 167
 - get_apple_remote() (in module pyglet.input), 217
 - get_audio_driver() (in module pyglet.media.drivers), 222
 - get_buffer_manager() (in module pyglet.image), 201
 - get_cached_animation_names (in module pyglet.resource), 255
 - get_cached_image_names (in module pyglet.resource), 255
 - get_cached_texture_names (in module pyglet.resource), 256
 - get_current_context() (in module pyglet.gl), 138
 - get_decoder() (in module pyglet.text), 299
 - get_decoders() (in module pyglet.image.codecs), 167
 - get_default() (in module pyglet.clock), 104
 - get_devices() (in module pyglet.input), 217
 - get_display() (in module pyglet.canvas), 101
 - get_encoders() (in module pyglet.image.codecs), 168
 - get_extensions (in module pyglet.gl.gl_info), 132
 - get_extensions (in module pyglet.gl.glu_info), 135
 - get_fps() (in module pyglet.clock), 105
 - get_fps_limit() (in module pyglet.clock), 105
 - get_grapheme_clusters() (in module pyglet.font.base), 119
 - get_joysticks() (in module pyglet.input), 217
 - get_max_texture_size() (in module pyglet.image.atlas), 165
 - get_platform() (in module pyglet.window), 315
 - get_renderer (in module pyglet.gl.gl_info), 133
 - get_script_home() (in module pyglet.resource), 254
 - get_settings_path() (in module pyglet.resource), 254
 - get_silent_audio_driver() (in module pyglet.media.drivers), 222
 - get_sleep_time() (in module pyglet.clock), 105
 - get_source_loader() (in module pyglet.media.sources.loader), 234
 - get_tablets() (in module pyglet.input), 217
 - get_texture_bins (in module pyglet.resource), 256
 - get_vendor (in module pyglet.gl.gl_info), 133
 - get_version (in module pyglet.gl.gl_info), 133
 - get_version (in module pyglet.gl.glu_info), 135
 - gl (in module pyglet.canvas.base), 99
 - gl (in module pyglet.font), 125
 - gl (in module pyglet.gl.base), 131
 - GLException, 136
 - GLInfo (class in pyglet.gl.gl_info), 132
 - GLUIInfo (class in pyglet.gl.glu_info), 134
 - Glyph (class in pyglet.font.base), 113
 - GlyphRenderer (class in pyglet.font.base), 116
 - GlyphString (class in pyglet.font.text), 121
 - GlyphTextureAtlas (class in pyglet.font.base), 116
 - Group (class in pyglet.graphics), 158
 - group (Sprite attribute), 260
- ## H
- halign (Text attribute), 122
 - has_exit (EventLoop attribute), 91
 - has_exit (Window attribute), 313
 - has_exit (WindowExitHandler attribute), 303
 - HAT (AbsoluteAxis attribute), 206
 - HAT_X (AbsoluteAxis attribute), 206
 - HAT_Y (AbsoluteAxis attribute), 206
 - have_avbin() (in module pyglet.media.sources.loader), 234
 - have_context (GLInfo attribute), 132
 - have_context (GLUIInfo attribute), 134
 - have_context() (in module pyglet.gl.gl_info), 132
 - have_extension (in module pyglet.gl.gl_info), 133
 - have_extension (in module pyglet.gl.glu_info), 135
 - have_font() (in module pyglet.font), 125
 - have_font() (pyglet.font.base.Font class method), 113
 - have_version (in module pyglet.gl.gl_info), 133
 - have_version (in module pyglet.gl.glu_info), 135
 - header_fmt (RIFFChunk attribute), 245
 - header_length (RIFFChunk attribute), 245

height (IncrementalTextLayout attribute), 278
 height (Screen attribute), 98
 height (ScreenMode attribute), 99
 height (ScrollableTextLayout attribute), 280
 height (ScrollableTextLayoutGroup attribute), 282
 height (Sprite attribute), 260
 height (Text attribute), 122
 height (TextLayout attribute), 284
 height (Window attribute), 313
 html (in module pyglet.resource), 256
 HTMLDecoder (class in pyglet.text.formats.html), 272
 HTMLLabel (class in pyglet.text), 293

I

IBM_FORMAT_ADPCM (in module
 glet.media.sources.riff), 249
 IBM_FORMAT_ALAW (in module
 glet.media.sources.riff), 249
 IBM_FORMAT_MULAW (in module
 glet.media.sources.riff), 249
 image (in module pyglet.resource), 256
 image (Sprite attribute), 260
 image_data (AbstractImage attribute), 169
 image_data (Texture attribute), 187
 ImageData (class in pyglet.image), 180
 ImageDataRegion (class in pyglet.image), 182
 ImageDecodeException, 166
 ImageDecoder (class in pyglet.image.codecs), 166
 ImageElement (class in pyglet.text.formats.structured),
 273
 ImageEncodeException, 167
 ImageEncoder (class in pyglet.image.codecs), 166
 ImageException, 201
 ImageGrid (class in pyglet.image), 183
 ImageMouseCursor (class in pyglet.window), 309
 ImagePattern (class in pyglet.image), 184
 images (Texture attribute), 187
 IncrementalTextLayout (class in pyglet.text.layout), 277
 IndexedVertexDomain (class in
 py-glet.graphics.vertexdomain), 154
 IndexedVertexList (class in
 py-glet.graphics.vertexdomain), 154
 indices (IndexedVertexList attribute), 155
 IndirectArrayRegion (class in
 py-glet.graphics.vertexbuffer), 150
 info (Source attribute), 229
 InlineElement (class in pyglet.text.document), 269
 interleave_attributes() (in module
 py-glet.graphics.vertexattribute), 147
 invalid (Window attribute), 313
 is_queued (StreamingSource attribute), 232
 italic (DocumentLabel attribute), 291
 item_height (Texture3D attribute), 188
 item_height (TextureGrid attribute), 192

item_height (UniformTextureSequence attribute), 200
 item_width (Texture3D attribute), 188
 item_width (TextureGrid attribute), 192
 item_width (UniformTextureSequence attribute), 200
 items (Texture3D attribute), 188
 items (TextureGrid attribute), 192

J

Joystick (class in pyglet.input.base), 211

K

KeyStateHandler (class in pyglet.window.key), 304

L

py-Label (class in pyglet.text), 295
 py-leading (Text attribute), 122
 py-LEFT (in module pyglet.window.mouse), 307
 py-left (ScrollableTextLayoutGroup attribute), 282
 py-LEFT (Text attribute), 122
 level (Texture attribute), 187
 line (Caret attribute), 264
 line_height (GlyphTextureAtlas attribute), 117
 line_height (Text attribute), 122
 LinearDecayEnvelope (class in
 py-glet.media.sources.procedural), 238
 link_AGL (in module pyglet.gl.lib), 137
 link_WGL (in module pyglet.gl.lib), 137
 ListBuilder (class in pyglet.text.formats.structured), 274
 load() (in module pyglet.font), 125
 load() (in module pyglet.image), 202
 load() (in module pyglet.media.sources.loader), 234
 load() (in module pyglet.text), 300
 load_animation() (in module pyglet.image), 202
 Loader (class in pyglet.resource), 252
 Location (class in pyglet.resource), 253
 location (in module pyglet.resource), 256
 loop (SourceGroup attribute), 229

M

major_version (Config attribute), 130
 MappableVertexBufferObject (class in
 py-glet.graphics.vertexbuffer), 150
 mark (Caret attribute), 264
 max_distance (Player attribute), 225
 media (in module pyglet.resource), 256
 MediaEvent (class in pyglet.media.events), 222
 MediaException, 223
 MediaFormatException, 223
 MediaThread (class in pyglet.media.threads), 250
 MIDDLE (in module pyglet.window.mouse), 307
 min_distance (Player attribute), 225
 MIN_SLEEP (Clock attribute), 104
 minor_version (Config attribute), 130

mipmapped_texture (AbstractImage attribute), 169
 missing_function() (in module pyglet.gl.lib), 137
 MissingFunctionException, 136
 modifiers_string() (in module pyglet.window.key), 305
 motion_string() (in module pyglet.window.key), 305
 MouseCursor (class in pyglet.window), 309
 MouseCursorException, 314
 multi_tex_coords (VertexList attribute), 156
 multiline (IncrementalTextLayout attribute), 278
 multiline (TextLayout attribute), 284
 MultiTexCoordAttribute (class in pyglet.graphics.vertexattribute), 145

N

name (Display attribute), 97
 NormalAttribute (class in pyglet.graphics.vertexattribute), 145
 normals (VertexList attribute), 156
 NoSuchConfigException, 314
 NoSuchDisplayException, 314
 NoSuchScreenModeException, 315
 null_group (in module pyglet.graphics), 161
 NullGroup (class in pyglet.graphics), 159

O

ObjectSpace (class in pyglet.gl.base), 130
 opacity (Sprite attribute), 260
 options (in module pyglet), 79
 OrderedGroup (class in pyglet.graphics), 159
 OrderedListBuilder (class in pyglet.text.formats.structured), 274
 OverriddenRunIterator (class in pyglet.text.runlist), 288
 owner (BufferImage attribute), 172

P

path (in module pyglet.resource), 257
 PERIOD (Caret attribute), 264
 pitch (Player attribute), 226
 PlainTextDecoder (class in pyglet.text.formats.plaintext), 273
 Platform (class in pyglet.window), 310
 platform_event_loop (in module pyglet.app), 95
 PlatformEventLoop (class in pyglet.app.base), 91
 Player (class in pyglet.media.player), 225
 PlayerGroup (class in pyglet.media.player), 226
 PlayerWorker (class in pyglet.media.threads), 250
 playing (Player attribute), 226
 plural (ColorAttribute attribute), 144
 plural (EdgeFlagAttribute attribute), 144
 plural (FogCoordAttribute attribute), 145
 plural (NormalAttribute attribute), 145
 plural (SecondaryColorAttribute attribute), 146
 plural (TexCoordAttribute attribute), 146
 plural (VertexAttribute attribute), 147

position (AbstractListener attribute), 224
 position (Caret attribute), 264
 position (InlineElement attribute), 269
 position (Player attribute), 226
 position (Sprite attribute), 260
 print_function (in module pyglet.app.base), 93
 print_function (in module pyglet.clock), 107
 print_function (in module pyglet.debug), 108
 print_function (in module pyglet.font.base), 120
 print_function (in module pyglet.font.text), 123
 print_function (in module pyglet.gl), 139
 print_function (in module pyglet.gl.lib), 137
 print_function (in module pyglet.graphics), 161
 print_function (in module pyglet.graphics.allocation), 142
 print_function (in module pyglet.graphics.vertexattribute), 148
 print_function (in module pyglet.graphics.vertexbuffer), 153
 print_function (in module pyglet.graphics.vertexdomain), 157
 print_function (in module pyglet.image), 202
 print_function (in module pyglet.info), 204
 print_function (in module pyglet.media.drivers), 222
 print_function (in module pyglet.media.drivers.silent), 221
 print_function (in module pyglet.media.player), 226
 print_function (in module pyglet.media.sources.base), 233
 print_function (in module pyglet.media.sources.loader), 235
 print_function (in module pyglet.media.threads), 250
 print_function (in module pyglet.sprite), 262
 print_function (in module pyglet.text.layout), 286
 print_function (in module pyglet.window.event), 303
 ProceduralSource (class in pyglet.media.sources.procedural), 238
 ptr (AbstractBuffer attribute), 149
 pyglet (module), 79, 89
 pyglet.app (module), 89
 pyglet.app.base (module), 90
 pyglet.canvas (module), 96
 pyglet.canvas.base (module), 96
 pyglet.clock (module), 102
 pyglet.debug (module), 107
 pyglet.event (module), 108
 pyglet.font (module), 111
 pyglet.font.base (module), 112
 pyglet.font.text (module), 120
 pyglet.font.ttf (module), 124
 pyglet.gl (module), 127
 pyglet.gl.base (module), 127
 pyglet.gl.gl (module), 131
 pyglet.gl.gl_info (module), 131

pyglet.gl.glu (module), 133
 pyglet.gl.glu_info (module), 134
 pyglet.gl.lib (module), 135
 pyglet.graphics (module), 139
 pyglet.graphics.allocation (module), 141
 pyglet.graphics.vertexattribute (module), 142
 pyglet.graphics.vertexbuffer (module), 148
 pyglet.graphics.vertexdomain (module), 154
 pyglet.image (module), 161
 pyglet.image.atlas (module), 163
 pyglet.image.codecs (module), 165
 pyglet.image.codecs.bmp (module), 165
 pyglet.image.codecs.dds (module), 165
 pyglet.image.codecs.gif (module), 165
 pyglet.image.codecs.png (module), 166
 pyglet.image.codecs.s3tc (module), 166
 pyglet.info (module), 203
 pyglet.input (module), 205
 pyglet.input.base (module), 205
 pyglet.input.evdev_constants (module), 216
 pyglet.media (module), 218
 pyglet.media.drivers (module), 219
 pyglet.media.drivers.base (module), 219
 pyglet.media.drivers.silent (module), 219
 pyglet.media.events (module), 222
 pyglet.media.exceptions (module), 222
 pyglet.media.listener (module), 223
 pyglet.media.player (module), 224
 pyglet.media.sources (module), 226
 pyglet.media.sources.base (module), 227
 pyglet.media.sources.loader (module), 233
 pyglet.media.sources.procedural (module), 235
 pyglet.media.sources.riff (module), 244
 pyglet.media.threads (module), 249
 pyglet.resource (module), 251
 pyglet.sprite (module), 257
 pyglet.text (module), 262
 pyglet.text.caret (module), 263
 pyglet.text.document (module), 265
 pyglet.text.formats (module), 270
 pyglet.text.formats.attributed (module), 271
 pyglet.text.formats.html (module), 271
 pyglet.text.formats.plaintext (module), 272
 pyglet.text.formats.structured (module), 273
 pyglet.text.layout (module), 275
 pyglet.text.runlist (module), 286
 pyglet.window (module), 300
 pyglet.window.event (module), 302
 pyglet.window.key (module), 303
 pyglet.window.mouse (module), 306
 pytest.mark.integration() (in module tests.annotations), 321
 pytest.mark.interactive() (in module tests.annotations), 321

pytest.mark.only_interactive() (in module tests.annotations), 322
 pytest.mark.requires_user_action() (in module tests.annotations), 321
 pytest.mark.requires_user_validation() (in module tests.annotations), 321
 pytest.mark.unit() (in module tests.annotations), 321

R

rate (ScreenMode attribute), 99
 register_event_type() (pyglet.event.EventDispatcher class method), 110
 reindex (in module pyglet.resource), 257
 RelativeAxis (class in pyglet.input.base), 213
 remove_active_context (in module pyglet.gl.gl_info), 133
 renderer (GLInfo attribute), 132
 require_gl_extension() (in module tests.annotations), 321
 require_platform() (in module tests.annotations), 321
 resizable (Window attribute), 313
 ResourceNotFoundException, 254
 RIFFChunk (class in pyglet.media.sources.riff), 245
 RIFFFile (class in pyglet.media.sources.riff), 245
 RIFFForm (class in pyglet.media.sources.riff), 245
 RIFFFormatException, 248
 RIFFSourceLoader (class in pyglet.media.sources.loader), 234
 RIFFType (class in pyglet.media.sources.riff), 246
 RIGHT (in module pyglet.window.mouse), 307
 RIGHT (Text attribute), 122
 rotation (Sprite attribute), 260
 rows (TextureGrid attribute), 192
 run() (in module pyglet.app), 94
 RunIterator (class in pyglet.text.runlist), 288
 RunList (class in pyglet.text.runlist), 289
 RX (AbsoluteAxis attribute), 207
 RX (RelativeAxis attribute), 213
 RY (AbsoluteAxis attribute), 207
 RY (RelativeAxis attribute), 213
 RZ (AbsoluteAxis attribute), 207
 RZ (RelativeAxis attribute), 213

S

Sawtooth (class in pyglet.media.sources.procedural), 239
 scale (Sprite attribute), 260
 scale_x (Sprite attribute), 260
 scale_y (Sprite attribute), 260
 schedule() (in module pyglet.clock), 105
 schedule_interval() (in module pyglet.clock), 105
 schedule_interval_soft() (in module pyglet.clock), 106
 schedule_once() (in module pyglet.clock), 106
 Screen (class in pyglet.canvas.base), 97
 screen (Window attribute), 313
 ScreenMode (class in pyglet.canvas.base), 98
 SCROLL_INCREMENT (Caret attribute), 264

ScrollableTextLayout (class in `pyglet.text.layout`), 279

ScrollableTextLayoutGroup (class in `pyglet.text.layout`), 281

secondary_colors (VertexList attribute), 156

SecondaryColorAttribute (class in `pyglet.graphics.vertexattribute`), 146

selection_background_color (IncrementalTextLayout attribute), 278

selection_color (IncrementalTextLayout attribute), 278

selection_end (IncrementalTextLayout attribute), 278

selection_start (IncrementalTextLayout attribute), 278

serialize_attributes() (in module `pyglet.graphics.vertexattribute`), 147

set_active_context (in module `pyglet.gl.gl_info`), 133

set_active_context (in module `pyglet.gl.glu_info`), 135

set_default() (in module `pyglet.clock`), 106

set_fps_limit() (in module `pyglet.clock`), 106

Silence (class in `pyglet.media.sources.procedural`), 240

SilentAudioBuffer (class in `pyglet.media.drivers.silent`), 220

SilentAudioDriver (class in `pyglet.media.drivers.silent`), 220

SilentAudioPacket (class in `pyglet.media.drivers.silent`), 221

SilentAudioPlayerPacketConsumer (class in `pyglet.media.drivers.silent`), 221

SilentTimeAudioPlayer (class in `pyglet.media.drivers.silent`), 221

Sine (class in `pyglet.media.sources.procedural`), 241

size (AbstractBuffer attribute), 149

skip_platform() (in module `tests.annotations`), 321

SLEEP_UNDERSHOOT (Clock attribute), 104

SolidColorImagePattern (class in `pyglet.image`), 185

Source (class in `pyglet.media.sources.base`), 228

source (Player attribute), 226

SourceGroup (class in `pyglet.media.sources.base`), 229

SourceInfo (class in `pyglet.media.sources.base`), 229

Sprite (class in `pyglet.sprite`), 259

SpriteGroup (class in `pyglet.sprite`), 261

Square (class in `pyglet.media.sources.procedural`), 242

StaticMemorySource (class in `pyglet.media.sources.base`), 230

StaticSource (class in `pyglet.media.sources.base`), 231

StreamingSource (class in `pyglet.media.sources.base`), 232

StructuredTextDecoder (class in `pyglet.text.formats.structured`), 274

style (Window attribute), 313

STYLE_INDETERMINATE (in module `pyglet.text.document`), 270

symbol_string() (in module `pyglet.window.key`), 306

TabletCanvas (class in `pyglet.input.base`), 214

TabletCursor (class in `pyglet.input.base`), 215

test_clock() (in module `pyglet.clock`), 106

tex_coords (Texture attribute), 187

tex_coords (VertexList attribute), 156

tex_coords_order (Texture attribute), 187

TexCoordAttribute (class in `pyglet.graphics.vertexattribute`), 146

text (AbstractDocument attribute), 267

Text (class in `pyglet.font.text`), 121

text (DocumentLabel attribute), 291

text (HTMLLabel attribute), 293

text (in module `pyglet.resource`), 257

text (Text attribute), 123

TextLayout (class in `pyglet.text.layout`), 282

TextLayoutForegroundDecorationGroup (class in `pyglet.text.layout`), 285

TextLayoutForegroundGroup (class in `pyglet.text.layout`), 285

TextLayoutGroup (class in `pyglet.text.layout`), 285

TextLayoutTextureGroup (class in `pyglet.text.layout`), 286

texture (AbstractImage attribute), 169

Texture (class in `pyglet.image`), 185

texture (in module `pyglet.resource`), 257

Texture3D (class in `pyglet.image`), 188

texture_height (Font attribute), 113

texture_internalformat (Font attribute), 113

texture_sequence (AbstractImageSequence attribute), 170

texture_width (Font attribute), 113

TextureAtlas (class in `pyglet.image.atlas`), 164

TextureBin (class in `pyglet.image.atlas`), 164

TextureGrid (class in `pyglet.image`), 191

TextureGroup (class in `pyglet.graphics`), 159

TextureRegion (class in `pyglet.image`), 194

TextureSequence (class in `pyglet.image`), 197

tick() (in module `pyglet.clock`), 107

TileableTexture (class in `pyglet.image`), 197

time (Player attribute), 226

title (SourceInfo attribute), 230

top (ScrollableTextLayoutGroup attribute), 282

TOP (Text attribute), 122

top_group (TextLayout attribute), 284

track (SourceInfo attribute), 230

translate_x (ScrollableTextLayoutGroup attribute), 282

translate_y (ScrollableTextLayoutGroup attribute), 282

Triangle (class in `pyglet.media.sources.procedural`), 243

TruetypeInfo (class in `pyglet.font.ttf`), 124

U

UnformattedDocument (class in `pyglet.text.document`), 269

UniformTextureSequence (class in `pyglet.image`), 200

T

Tablet (class in `pyglet.input.base`), 214

UnorderedListBuilder (class in
 glet.text.formats.structured), 275
 unschedule() (in module pyglet.clock), 107
 up_orientation (AbstractListener attribute), 224
 update_period (FPSDisplay attribute), 309
 URLLocation (class in pyglet.resource), 253
 user_key() (in module pyglet.window.key), 306

V

valign (Text attribute), 123
 value (Button attribute), 209
 value (Control attribute), 210
 value (RelativeAxis attribute), 213
 vendor (GLInfo attribute), 132
 version (GLInfo attribute), 132
 version (GLUInfo attribute), 134
 version (in module pyglet), 80
 vertex_list() (in module pyglet.graphics), 160
 vertex_list_indexed() (in module pyglet.graphics), 160
 VertexArray (class in pyglet.graphics.vertexbuffer), 151
 VertexArrayRegion (class in py-
 glet.graphics.vertexbuffer), 151
 VertexAttribute (class in pyglet.graphics.vertexattribute),
 146
 VertexBufferObject (class in py-
 glet.graphics.vertexbuffer), 152
 VertexBufferObjectRegion (class in py-
 glet.graphics.vertexbuffer), 152
 VertexDomain (class in pyglet.graphics.vertexdomain),
 155
 VertexList (class in pyglet.graphics.vertexdomain), 156
 vertices (Glyph attribute), 114
 vertices (VertexList attribute), 156
 video_format (Source attribute), 229
 VideoFormat (class in pyglet.media.sources.base), 233
 view_x (ScrollableTextLayout attribute), 280
 view_x (ScrollableTextLayoutGroup attribute), 282
 view_y (IncrementalTextLayout attribute), 278
 view_y (ScrollableTextLayout attribute), 280
 view_y (ScrollableTextLayoutGroup attribute), 282
 visible (Caret attribute), 265
 visible (Sprite attribute), 261
 visible (Window attribute), 313
 volume (AbstractListener attribute), 224
 volume (Player attribute), 226
 vsync (Window attribute), 313

W

WAVE_FORMAT_PCM (in module py-
 glet.media.sources.riff), 249
 WaveDataChunk (class in pyglet.media.sources.riff), 246
 WaveForm (class in pyglet.media.sources.riff), 247
 WaveFormatChunk (class in pyglet.media.sources.riff),
 247

py- WAVEFormatException, 248
 WaveSource (class in pyglet.media.sources.riff), 247
 WeakSet (class in pyglet.app), 94
 WHEEL (RelativeAxis attribute), 213
 WhiteNoise (class in pyglet.media.sources.procedural),
 243
 width (IncrementalTextLayout attribute), 278
 width (Screen attribute), 98
 width (ScreenMode attribute), 99
 width (ScrollableTextLayout attribute), 280
 width (ScrollableTextLayoutGroup attribute), 282
 width (Sprite attribute), 261
 width (Text attribute), 123
 width (TextLayout attribute), 284
 width (Window attribute), 313
 Window (class in pyglet.window), 310
 window (in module pyglet.canvas.base), 100
 WINDOW_STYLE_BORDERLESS (Window attribute),
 312
 WINDOW_STYLE_DEFAULT (Window attribute), 312
 WINDOW_STYLE_DIALOG (Window attribute), 312
 WINDOW_STYLE_TOOL (Window attribute), 312
 WindowEventLogger (class in pyglet.window.event), 302
 WindowException, 315
 WindowExitHandler (class in pyglet.window.event), 302
 windows (in module pyglet.app), 95
 WorkerThread (class in pyglet.media.threads), 250

X

X (AbsoluteAxis attribute), 207
 x (GlyphTextureAtlas attribute), 117
 X (RelativeAxis attribute), 213
 x (Screen attribute), 98
 x (ScrollableTextLayout attribute), 280
 x (Sprite attribute), 261
 x (Text attribute), 123
 x (TextLayout attribute), 285
 x (Texture attribute), 187
 x_screen (Display attribute), 97

Y

Y (AbsoluteAxis attribute), 207
 y (GlyphTextureAtlas attribute), 117
 Y (RelativeAxis attribute), 213
 y (Screen attribute), 98
 y (ScrollableTextLayout attribute), 280
 y (Sprite attribute), 261
 y (Text attribute), 123
 y (TextLayout attribute), 285
 y (Texture attribute), 187
 year (SourceInfo attribute), 230

Z

Z (AbsoluteAxis attribute), 207

[Z \(RelativeAxis attribute\)](#), [213](#)

[z \(Text attribute\)](#), [123](#)

[z \(Texture attribute\)](#), [187](#)

[ZIPLocation \(class in pyglet.resource\)](#), [253](#)

[ZipRunIterator \(class in pyglet.text.runlist\)](#), [289](#)