Contents

1 Programming Guide 3
  1.1 pyglet Programming Guide ....................................................... 3

2 API Reference 87
  2.1 pyglet ................................................................. 87

3 Internals 449
  3.1 Importing pyglet .......................................................... 449
  3.2 Advanced topics .......................................................... 450
  3.3 OpenGL Interface Implementation ....................................... 451
  3.4 ctypes Wrapper Generation ............................................... 452
  3.5 wraptypes .............................................................. 453
  3.6 Making a pyglet release .................................................. 457
  3.7 Distribution ............................................................ 458
  3.8 Documentation .......................................................... 459
  3.9 tests.test ................................................................. 464

4 Related Documentation 469

Python Module Index 471
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.
1.1 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.

1.1.1 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
eexport 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
```

### 1.1.2 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`:

```python
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:

```python
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:

```python
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 file:

```python
import pyglet

music = pyglet.resource.media('music.mp3')
music.play()
pyglet.app.run()
```

---

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.
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:

```python
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.

### 1.1.3 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:

```python
>>> 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:

```python
>>> 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:

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

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

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

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.

---

8 Chapter 1. Programming Guide
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:

```python
>>> 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 `display.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 `display.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 auxiliary buffer is configured the same as the colour buffer. Up to four auxiliary buffers can typically be created. Specify 0 if you do not require any auxiliary buffers.

Like the accumulation buffer, auxiliary 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>double_buffer</td>
<td>True</td>
</tr>
<tr>
<td>depth_size</td>
<td>24</td>
</tr>
</tbody>
</table>
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:

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

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:

```python
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)
win = 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:

```python
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)
win = 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 auxilliary buffer or an accumulation buffer are printed:

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

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

1.1.4 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:

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

```python
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():
    glClearColor(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`:

```python
@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)

pyglet.app.run()
```

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:

```python
@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 GLException 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:

```python
# 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:

```python
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:

```python
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:

```python
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 default. 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.
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:

```python
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.wglext_arb` or `pyglet.gl.wglext_nv`.

1.1.5 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):

```python
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.

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:

```python
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):

```python
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:

```python
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).

**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", "v3f", "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.

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

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

<table>
<thead>
<tr>
<th>Format</th>
<th>Type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;b&quot;</td>
<td>Signed byte</td>
<td>int</td>
</tr>
<tr>
<td>&quot;B&quot;</td>
<td>Unsigned byte</td>
<td>int</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>Signed short</td>
<td>int</td>
</tr>
<tr>
<td>&quot;S&quot;</td>
<td>Unsigned short</td>
<td>int</td>
</tr>
<tr>
<td>&quot;i&quot;</td>
<td>Signed int</td>
<td>int</td>
</tr>
<tr>
<td>&quot;I&quot;</td>
<td>Unsigned int</td>
<td>int</td>
</tr>
<tr>
<td>&quot;f&quot;</td>
<td>Single precision float</td>
<td>float</td>
</tr>
<tr>
<td>&quot;d&quot;</td>
<td>Double precision float</td>
<td>float</td>
</tr>
</tbody>
</table>

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:
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:

```python
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:

```python
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.

<table>
<thead>
<tr>
<th>Vertex attribute</th>
<th>Object attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex position</td>
<td>vertices</td>
</tr>
<tr>
<td>Color</td>
<td>colors</td>
</tr>
<tr>
<td>Edge flag</td>
<td>edge_flags</td>
</tr>
<tr>
<td>Fog coordinate</td>
<td>fog_coords</td>
</tr>
<tr>
<td>Normal</td>
<td>normals</td>
</tr>
<tr>
<td>Secondary color</td>
<td>secondary_colors</td>
</tr>
<tr>
<td>Texture coordinate</td>
<td>tex_coords 4</td>
</tr>
<tr>
<td>Generic attribute</td>
<td>Inaccessible</td>
</tr>
</tbody>
</table>

---

4Only texture coordinates for texture unit 0 are accessible through this attribute.
In the following example, the vertex positions of the vertex list are updated by replacing the `vertices` attribute:

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

The attributes can also be selectively updated in-place:

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

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

```python
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:

```python
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:

<table>
<thead>
<tr>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;/static&quot;</td>
<td>Data is never or rarely modified after initialisation</td>
</tr>
<tr>
<td>&quot;/dynamic&quot;</td>
<td>Data is occasionally modified (default)</td>
</tr>
<tr>
<td>&quot;/stream&quot;</td>
<td>Data is updated every frame</td>
</tr>
</tbody>
</table>

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:

```python
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.

**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:

```python
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:

```python
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:

```python
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:

```python
from pyglet.gl import *

glEnable(texture.target)
glBindTexture(texture.target, texture.id)
```

before drawing vertex lists, and then:

```python
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:

```python
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:
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:

```python
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:

```python
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)
    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))
```

```python
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)))
```
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:

```python
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:

```python
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.

### 1.1.6 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

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

```python
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:

```python
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:
  ```python
c context = config.create_context(share)
c 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:
  ```python
c config = screen.get_best_config(template)
c 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:
  ```python
c config = gl.Config(double_buffer=True)
c 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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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.

<table>
<thead>
<tr>
<th>Style</th>
<th>Windows XP</th>
<th>Mac OS X</th>
</tr>
</thead>
<tbody>
<tr>
<td>WINDOW_STYLE_DEFAULT</td>
<td><img src="image" alt="Windows XP Default" /></td>
<td><img src="image" alt="Mac OS X Default" /></td>
</tr>
<tr>
<td>WINDOW_STYLE_DIALOG</td>
<td><img src="image" alt="Windows XP Dialog" /></td>
<td><img src="image" alt="Mac OS X Dialog" /></td>
</tr>
<tr>
<td>WINDOW_STYLE_TOOL</td>
<td><img src="image" alt="Windows XP Tool" /></td>
<td><img src="image" alt="Mac OS X Tool" /></td>
</tr>
</tbody>
</table>
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:

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

### 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:

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

### Icon

The window icon appears in the title bar and task bar 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:

```python
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:

```python
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:

```python
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.

1.1.7 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:
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`:

```python
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:

```python
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:

```python
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.

### 1.1.8 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:

```python
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:

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

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

While this technique is straightforward, 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:

```python
window = pyglet.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:

```python
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:

```python
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`:

```python
@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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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 observerable 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:

```python
# 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

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:

```python
import sys

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

            :event:
            '''
```

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

### 1.1.9 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:

```python
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 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:
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:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOD_SHIFT</td>
<td>Not available on Mac OS X</td>
</tr>
<tr>
<td>MOD_CTRL</td>
<td>Available on Windows only</td>
</tr>
<tr>
<td>MOD_ALT</td>
<td>Not available on Mac OS X</td>
</tr>
<tr>
<td>MOD_WINDOWS</td>
<td>Available on Windows only</td>
</tr>
<tr>
<td>MOD_COMMAND</td>
<td>Available on Mac OS X only</td>
</tr>
<tr>
<td>MOD_OPTION</td>
<td>Available on Mac OS X only</td>
</tr>
<tr>
<td>MOD_CAPSLOCK</td>
<td>Equivalent to MOD_CTRL, or MOD_COMMAND on Mac OS X.</td>
</tr>
<tr>
<td>MOD_NUMLOCK</td>
<td></td>
</tr>
<tr>
<td>MOD_SCROLLLOCK</td>
<td></td>
</tr>
<tr>
<td>MOD_ACCEL</td>
<td>Equivalent to MOD_CTRL, or MOD_COMMAND on Mac OS X.</td>
</tr>
</tbody>
</table>

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

```python
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:

```python
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:

```python
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:

```python
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.
### 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.

### 1.1.10 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

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).

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

```python
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:

```python
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:
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:

```python
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:

```python
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:

```python
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`:

```python
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`:

```python
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.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Windows XP</th>
<th>Mac OS X</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURSOR_DEFAULT</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_CROSSHAIR</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_HAND</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_HELP</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_NO</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_SIZE</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_SIZE_DOWN</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_SIZE_DOWN_LEFT</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_SIZE_DOWN_RIGHT</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_SIZE_LEFT</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_SIZE_LEFT_RIGHT</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_SIZE_RIGHT</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_SIZE_UP</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_SIZE_UP_DOWN</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_SIZE_UP_LEFT</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_SIZE_UP_RIGHT</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_TEXT</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_WAIT</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
<tr>
<td>CURSOR_WAIT_ARROW</td>
<td><img src="image" alt="cursor_icon" /></td>
<td><img src="image" alt="cursor_icon" /></td>
</tr>
</tbody>
</table>

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:

```python
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 from 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`:

```python
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.

1.1.11 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`:

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

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

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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`:

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

Then open it:

```python
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:

```python
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:

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

### 1.1.12 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:

```python
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:

```python
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`:

```python
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`:

```python
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:

```python
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.

<table>
<thead>
<tr>
<th>Animation parameter</th>
<th>Distance</th>
<th>Velocity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotation</td>
<td>Degrees</td>
<td>Degrees per second</td>
</tr>
<tr>
<td>Position</td>
<td>Pixels</td>
<td>Pixels per second</td>
</tr>
<tr>
<td>Keyframes</td>
<td>Frame number</td>
<td>Frames per second</td>
</tr>
</tbody>
</table>
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.

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

```python
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:

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

1.1. pyglet Programming Guide
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:

```python
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:

```python
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 remove_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:

```python
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:

```python
# 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 prepared 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 CopyrightAndFeelট©.

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:

```python
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:

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

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

```python
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:

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

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

```python
# 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):

```python
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:

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

These measurements are shown in the diagram below.

![Font metrics](image)

**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:

```python
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:

```python
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:

```python
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:

```python
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 all fully supported.

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

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

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.

---

[^3]: 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:

```python
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.

1.1.14 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

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

```python
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:

```python
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:

<table>
<thead>
<tr>
<th>Module</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pyglet.image.codecs.dds</code></td>
<td><code>DDSImageDecoder</code></td>
<td>Reads Microsoft DirectDraw Surface files containing compressed textures</td>
</tr>
<tr>
<td><code>pyglet.image.codecs.gdiplus</code></td>
<td><code>GDIPlusDecoder</code></td>
<td>Uses Windows GDI+ services to decode images.</td>
</tr>
<tr>
<td><code>pyglet.image.codecs.gdkpixbuf2</code></td>
<td><code>GdkPixbuf2ImageDecoder</code></td>
<td>Uses the GTK-2.0 GDK functions to decode images.</td>
</tr>
<tr>
<td><code>pyglet.image.codecs.pil</code></td>
<td><code>PILImageDecoder</code></td>
<td>Wrapper interface around PIL Image class.</td>
</tr>
<tr>
<td><code>pyglet.image.codecs.png</code></td>
<td><code>PNGImageDecoder</code></td>
<td>PNG decoder written in pure Python.</td>
</tr>
<tr>
<td><code>pyglet.image.codecs.quicktime</code></td>
<td><code>QuickTimeImageDecoder</code></td>
<td>Uses Mac OS X QuickTime to decode images.</td>
</tr>
</tbody>
</table>
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:

```python
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.

<table>
<thead>
<tr>
<th>Extension</th>
<th>Description</th>
<th>Windows XP</th>
<th>Mac OS X</th>
<th>Linux [6]</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bmp</td>
<td>Windows Bitmap</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>.dds</td>
<td>Microsoft DirectDraw Surface [7]</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>.exif</td>
<td>Exif</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.gif</td>
<td>Graphics Interchange Format</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>.jpg</td>
<td>JPEG/JIFF Image</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>.jpeg</td>
<td>JPEG/JIFF Image</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>.jp2</td>
<td>JPEG 2000</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>.jpx</td>
<td>PC Paintbrush Bitmap Graphic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.pcx</td>
<td>PC Paintbrush Bitmap Graphic</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.png</td>
<td>Portable Network Graphic</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>.pnm</td>
<td>PBM Portable Any Map Graphic Bitmap</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>.ras</td>
<td>Sun raster graphic</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.tga</td>
<td>Truevision Targa Graphic</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.tif</td>
<td>Tagged Image File Format</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>.tiff</td>
<td>Tagged Image File Format</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>.xbm</td>
<td>X11 bitmap</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>.xpm</td>
<td>X11 icon</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

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

---

[6] Requires GTK 2.0 or later.
[7] Only 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:

```python
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.](image)

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:

```python
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:

<table>
<thead>
<tr>
<th>Original class</th>
<th><code>get_texture()</code></th>
<th><code>get_image_data()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Texture</td>
<td>No change</td>
<td><code>glGetTexImage2D</code></td>
</tr>
<tr>
<td>TextureRegion</td>
<td>No change</td>
<td><code>glGetTexImage2D</code>, crop resulting image.</td>
</tr>
<tr>
<td>ImageData</td>
<td><code>glTexImage2D</code>¹</td>
<td>No change</td>
</tr>
<tr>
<td>ImageDataRegion</td>
<td><code>glTexImage2D</code>¹</td>
<td>No change</td>
</tr>
<tr>
<td>CompressedImageData</td>
<td><code>glCompressedTexImage2D</code>⁹</td>
<td>N/A ¹⁰</td>
</tr>
<tr>
<td>BufferImage</td>
<td><code>glCopyTexImage2D</code>¹¹</td>
<td><code>glReadPixels</code></td>
</tr>
</tbody>
</table>

¹¹`ImageData` caches the texture for future use, so there is no performance penalty for repeatedly blitting an `ImageData`.  
¹⁰N/A means no conversion is necessary.

---

1.1. pyglet Programming Guide 61
You should try to avoid conversions which use \texttt{glGetTexImage2D} or \texttt{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 \texttt{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 \texttt{ImageData} object for any image with \texttt{get_image_data}:

\begin{verbatim}
kitten = pyglet.image.load('kitten.png').get_image_data()
\end{verbatim}

The design of \texttt{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 \texttt{pitch} and \texttt{format} properties determine how the bytes are arranged. \texttt{pitch} gives the number of bytes between each consecutive row. The data is assumed to run from left-to-right, bottom-to-top, unless \texttt{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 \texttt{pitch} values are often used to align each row to machine word boundaries.

The \texttt{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 \( \text{len(kitten.format)} \times \text{kitten.width} \).

To retrieve pixel data in a particular format, use the \texttt{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):

\begin{verbatim}
kitten = kitten.get_image_data()
data = kitten.get_data('RGB', kitten.width * 3)
\end{verbatim}

\texttt{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 \texttt{set_data}, again specifying the format and pitch:

\begin{verbatim}
kitten.set_data('RGB', kitten.width * 3, data)
\end{verbatim}

You can also create \texttt{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.

\footnote{If the required texture compression extension is not present, the image is decompressed in memory and then supplied to OpenGL via \texttt{glTexImage2D}.}

\footnote{It is not currently possible to retrieve \texttt{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., \texttt{compressed_image.get_texture().get_image_data()}.}

\footnote{\texttt{BufferImageMask} cannot be converted to \texttt{Texture}.}
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).

<table>
<thead>
<tr>
<th>Format</th>
<th>Required extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;I&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;L&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;LA&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;R&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;G&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;B&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;RGB&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;RGBA&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;ARGB&quot;</td>
<td>GL_EXT_bgra and GL_APPLE_packed_pixels</td>
</tr>
<tr>
<td>&quot;ABGR&quot;</td>
<td>GL_EXT_abgr</td>
</tr>
<tr>
<td>&quot;BGR&quot;</td>
<td>GL_EXT_bgra</td>
</tr>
<tr>
<td>&quot;BGRA&quot;</td>
<td>GL_EXT_bgra</td>
</tr>
</tbody>
</table>

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.

![Image](image.png)

**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.

This image has one row and eight columns. This is all the information you need to create an *ImageGrid* with:
Fig. 1.8: An image consisting of eight animation frames arranged in a grid.

```python
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:

```python
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:

```python
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`:

```python
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:

```python
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:

```python
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:

```python
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()
```
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.](image)

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:

```py
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):

```py
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:

```py
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:

```py
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:
When using the stencil or auxiliary 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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
>>> 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:

```python
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.

<table>
<thead>
<tr>
<th>Format</th>
<th>Internal format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any format with 3 components</td>
<td>GL_RGB</td>
</tr>
<tr>
<td>Any format with 2 components</td>
<td>GL_LUMINANCE_ALPHA</td>
</tr>
<tr>
<td>&quot;A&quot;</td>
<td>GL_ALPHA</td>
</tr>
<tr>
<td>&quot;L&quot;</td>
<td>GL_LUMINANCE</td>
</tr>
<tr>
<td>&quot;I&quot;</td>
<td>GL_INTENSITY</td>
</tr>
<tr>
<td>Any other format</td>
<td>GL_RGBA</td>
</tr>
</tbody>
</table>

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:

```python
kitten.save('kitten.png')
```

or, specifying a file-like object:

```python
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:

```python
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.

**1.1.15 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**
• **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:

<table>
<thead>
<tr>
<th>Windows</th>
<th>Mac OS X</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenAL (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DirectSound</td>
<td></td>
<td>Pulseaudio</td>
</tr>
</tbody>
</table>

The audio driver can be set through the `audio` key of the `pyglet.options` dictionary. For example:

```python
cpyget.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:

<table>
<thead>
<tr>
<th>String</th>
<th>Audio driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>openal</td>
<td>OpenAL</td>
</tr>
<tr>
<td>directsound</td>
<td>DirectSound</td>
</tr>
<tr>
<td>pulse</td>
<td>Pulseaudio</td>
</tr>
<tr>
<td>silent</td>
<td>No audio output</td>
</tr>
</tbody>
</table>

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.

---

12OpenAL 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
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 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:

```python
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 filename:

```python
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>width, height</td>
<td>Width and height of the video image, in pixels.</td>
</tr>
<tr>
<td>sample_aspect</td>
<td>The aspect ratio of each video pixel.</td>
</tr>
</tbody>
</table>

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:

```python
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.

1.1. pyglet Programming Guide
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:

```python
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`:

```python
explosion = pyglet.media.StaticSource(pyglet.media.load('explosion.wav'))
```

### 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:

```python
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:

```python
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:

```python
player.queue(source1)
player.queue(source2)
```

Playback begins with the player’s `play` method is called:

```python
player.play()
```

Standard controls for controlling playback are provided by these methods:
Method | Description
--- | ---
**play** | Begin or resume playback of the current source.
**pause** | Pause playback of the current source.
**next_source** | Dequeue the current source and move to the next one immediately. **next** can also be used but it is deprecated because of incompatibilities with Python 3.
**seek** | 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:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>time</em></td>
<td>The current playback position within the current source, in seconds. This is read-only (but see the <em>seek</em> method).</td>
</tr>
<tr>
<td><em>playing</em></td>
<td>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 <em>pause</em> and <em>play</em> methods).</td>
</tr>
<tr>
<td><em>source</em></td>
<td>A reference to the current source being played. This is read-only (but see the <em>queue</em> method).</td>
</tr>
<tr>
<td><em>volume</em></td>
<td>The audio level, expressed as a float from 0 (mute) to 1 (normal volume). This can be set at any time.</td>
</tr>
</tbody>
</table>

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:

```python
@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:
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.

### 1.1.16 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:

```python
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:

```python
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:

```python
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:

```python
data_file = open('file.txt')
```

use:

```python
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.
<table>
<thead>
<tr>
<th>File function</th>
<th>Resource function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>pyglet.resource.file</td>
<td>File-like object</td>
</tr>
<tr>
<td>pyglet.image.load</td>
<td>pyglet.resource.image</td>
<td>Texture or TextureRegion</td>
</tr>
<tr>
<td>pyglet.image.load</td>
<td>pyglet.resource.texture</td>
<td>Texture</td>
</tr>
<tr>
<td>pyglet.image.load_animation</td>
<td>pyglet.resource.animation</td>
<td>Animation</td>
</tr>
<tr>
<td>pyglet.media.load</td>
<td>pyglet.resource.media</td>
<td>Source</td>
</tr>
<tr>
<td>pyglet.text.load</td>
<td>pyglet.resource.text</td>
<td>UnformattedDocument</td>
</tr>
<tr>
<td>pyglet.text.load</td>
<td>pyglet.resource.html</td>
<td>FormattedDocument</td>
</tr>
<tr>
<td>pyglet.text.load</td>
<td>pyglet.resource.attributed</td>
<td>FormattedDocument</td>
</tr>
<tr>
<td>pyglet.font.add_file</td>
<td>pyglet.resource.add_font</td>
<td>None</td>
</tr>
</tbody>
</table>

`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 `<img src="image.png" />` 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:

```python
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:

```python
tyget.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:

```python
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:

```python
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:

```python
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:

```python
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/ApplicationName/ on Linux (depends on XDG_CONFIG_HOME environment variable).
• $HOME\Application Settings\ApplicationName on Windows
• ~/Library/Application Support/ApplicationName 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:

```python
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')
```

### 1.1.17 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:

```python
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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Environment variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug_font</td>
<td>PYGLET_DEBUG_FONT</td>
<td>bool</td>
</tr>
<tr>
<td>debug_gl</td>
<td>PYGLET_DEBUG_GL</td>
<td>bool</td>
</tr>
<tr>
<td>debug_gl_trace</td>
<td>PYGLET_DEBUG_GL_TRACE</td>
<td>bool</td>
</tr>
<tr>
<td>debug_gl_trace_args</td>
<td>PYGLET_DEBUG_GL_TRACE_ARGS</td>
<td>bool</td>
</tr>
<tr>
<td>debug_graphics_batch</td>
<td>PYGLET_DEBUG_GRAPHICS_BATCH</td>
<td>bool</td>
</tr>
<tr>
<td>debug_lib</td>
<td>PYGLET_DEBUG_LIB</td>
<td>bool</td>
</tr>
<tr>
<td>debug_media</td>
<td>PYGLET_DEBUG_MEDIA</td>
<td>bool</td>
</tr>
<tr>
<td>debug_trace</td>
<td>PYGLET_DEBUG_TRACE</td>
<td>bool</td>
</tr>
<tr>
<td>debug_trace_args</td>
<td>PYGLET_DEBUG_TRACE_ARGS</td>
<td>bool</td>
</tr>
<tr>
<td>debug_trace_depth</td>
<td>PYGLET_DEBUG_TRACE_DEPTH</td>
<td>int</td>
</tr>
<tr>
<td>debug_win32</td>
<td>PYGLET_DEBUG_WIN32</td>
<td>bool</td>
</tr>
<tr>
<td>debug_x11</td>
<td>PYGLET_DEBUG_X11</td>
<td>bool</td>
</tr>
<tr>
<td>graphics_vbo</td>
<td>PYGLET_GRAPHICS_VBO</td>
<td>bool</td>
</tr>
</tbody>
</table>

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.

1.1.18 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 superceded 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:

```python
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:

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

#### Application event loop

Every application using *pyglet* 1.0 provides its own event loop, such as:

```python
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:

```python
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):

```python
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`:

```python
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:

```python
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:

```python
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:

```python
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*:

```python
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):

```python
@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.
2.1 pyglet

pyglet is a cross-platform games and multimedia package.
Detailed documentation is available at http://www.pyglet.org

2.1.1 Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>Application-wide functionality.</td>
</tr>
<tr>
<td>canvas</td>
<td>Display and screen management.</td>
</tr>
<tr>
<td>clock</td>
<td>Precise framerate calculation, scheduling and framerate limiting.</td>
</tr>
<tr>
<td>event</td>
<td>Event dispatch framework.</td>
</tr>
<tr>
<td>extlibs</td>
<td>External dependencies for Pyglet.</td>
</tr>
<tr>
<td>font</td>
<td>Load fonts and render text.</td>
</tr>
<tr>
<td>gl</td>
<td>OpenGL and GLU interface.</td>
</tr>
<tr>
<td>graphics</td>
<td>Low-level graphics rendering.</td>
</tr>
<tr>
<td>image</td>
<td>Image load, capture and high-level texture functions.</td>
</tr>
<tr>
<td>info</td>
<td>Get environment information useful for debugging.</td>
</tr>
<tr>
<td>input</td>
<td>Joystick, tablet and USB HID device support.</td>
</tr>
<tr>
<td>media</td>
<td>Audio and video playback.</td>
</tr>
<tr>
<td>resource</td>
<td>Load application resources from a known path.</td>
</tr>
<tr>
<td>sprite</td>
<td>Display positioned, scaled and rotated images.</td>
</tr>
<tr>
<td>text</td>
<td>Text formatting, layout and display.</td>
</tr>
<tr>
<td>window</td>
<td>Windowing and user-interface events.</td>
</tr>
</tbody>
</table>

**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.

Classes

WeakSet  Set of objects, referenced weakly.

    WeakSet
    
    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.

    Constructor:
    __init__()

    Methods:

    add(value)
    remove(value)

Methods
WeakSet.add(value)
WeakSet.remove(value)

Exceptions

AppException

Exception defined in pyglet.app

exception AppException

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit()</td>
<td>Exit the application event loop.</td>
</tr>
<tr>
<td>run()</td>
<td>Begin processing events, scheduled functions and window updates.</td>
</tr>
</tbody>
</table>

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 = ‘linux2’

str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.
**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.

**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**
- `base`
- `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

### 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:

- base
- 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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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`:

```python
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:

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

Call draw on the ClockDisplay object for each frame:

```python
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:

```python
clk = clock.get_default()
```

You can also replace the default clock with your own:

```python
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

- **Clock**
  - Class for calculating and limiting framerate, and for calling scheduled functions.

- **ClockDisplay**
  - Display current clock values, such as FPS.

```python
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.

**Constructor:**

```python
__init__ (fps_limit=None, time_function=<built-in function time>)
```

Initialise a Clock, with optional framerate limit and custom time function.

**Parameters**

- **fps_limit** (float) – If not None, the maximum allowable framerate. Defaults to None. Deprecated in pyglet 1.2.

- **time_function** (function) – Function to return the elapsed time of the application, in seconds. Defaults to time.time, but can be replaced to allow for easy time dilation effects or game pausing.

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>call_scheduled_functions(dt)</code></td>
<td>Call scheduled functions that elapsed on the last <code>update_time</code>.</td>
</tr>
<tr>
<td><code>get_fps()</code></td>
<td>Get the average FPS of recent history.</td>
</tr>
<tr>
<td><code>get_fps_limit()</code></td>
<td>Get the framerate limit.</td>
</tr>
<tr>
<td><code>get_sleep_time(sleep_idle)</code></td>
<td>Get the time until the next item is scheduled.</td>
</tr>
<tr>
<td><code>schedule(func, *args, **kwargs)</code></td>
<td>Schedule a function to be called every frame.</td>
</tr>
<tr>
<td><code>schedule_interval(func, interval, *args, ...)</code></td>
<td>Schedule a function to be called every <code>interval</code> seconds.</td>
</tr>
</tbody>
</table>
Table 2.9 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>schedule_interval_soft</code></td>
<td>Schedule a function to be called every <code>interval</code> seconds, beginning at a time that does not coincide with other scheduled events.</td>
</tr>
<tr>
<td><code>schedule_once</code></td>
<td>Schedule a function to be called once after <code>delay</code> seconds.</td>
</tr>
<tr>
<td><code>set_fps_limit</code></td>
<td>Set the framerate limit.</td>
</tr>
<tr>
<td><code>sleep()</code></td>
<td>Signify that one frame has passed.</td>
</tr>
<tr>
<td><code>unschedule()</code></td>
<td>Remove a function from the schedule.</td>
</tr>
<tr>
<td><code>update_time()</code></td>
<td>Get the elapsed time since the last call to <code>update_time</code>.</td>
</tr>
</tbody>
</table>

Attributes:

- `MIN_SLEEP` The minimum amount of time in seconds this clock will attempt to sleep for when framerate limiting.
- `SLEEP_UNDERSHOOT` The amount of time in seconds this clock subtracts from sleep values to compensate for lazy operating systems.

Methods

- **Clock.** `call_scheduled_functions(dt)`
  Call scheduled functions that elapsed on the last `update_time`.
  
  **Note:** Since pyglet 1.2

  **Parameters** `dt (float)` – The elapsed time since the last update to pass to each scheduled function.
  This is *not* used to calculate which functions have elapsed.

  **Return type** `bool`

  **Returns** True if any functions were called, otherwise False.

- `get_fps()`
  Get the average FPS of recent history.
  The result is the average of a sliding window of the last “n” frames, where “n” is some number designed to cover approximately 1 second.

  **Return type** `float`

  **Returns** The measured frames per second.

- `get_fps_limit()`
  Get the framerate limit.

  **Return type** `float`

  **Returns** The framerate limit previously set in the constructor or `set_fps_limit`, or None if no limit was set.

- `get_sleep_time(sleep_idle)`
  Get the time until the next item is scheduled.
  This method considers all scheduled items and the current `fps_limit`, if any.
  Applications can choose to continue receiving updates at the maximum framerate during idle time (when no functions are scheduled), or they can sleep through their idle time and allow the CPU to switch to other processes or run in low-power mode.
If `sleep_idle` is `True` the latter behaviour is selected, and `None` will be returned if there are no scheduled items. Otherwise, if `sleep_idle` is `False`, a sleep time allowing the maximum possible framerate (considering `fps_limit`) will be returned; or an earlier time if a scheduled function is ready.

**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

```python
Clock.schedule (func, *args, **kwargs)
Schedule a function to be called every frame.

The function should have a prototype that includes `dt` as the first argument, which gives the elapsed time, in seconds, since the last clock tick. Any additional arguments given to this function are passed on to the callback:

```python
def callback (dt, *args, **kwargs):
    pass
```

**Parameters**

- `func (function)` — The function to call each frame.

```python
Clock.schedule_interval (func, interval, *args, **kwargs)
Schedule a function to be called every `interval` seconds.

Specifying an interval of 0 prevents the function from being called again (see `schedule` to call a function as often as possible).

The callback function prototype is the same as for `schedule`.

**Parameters**

- `func (function)` — The function to call when the timer lapses.
- `interval (float)` — The number of seconds to wait between each call.

```python
Clock.schedule_interval_soft (func, interval, *args, **kwargs)
Schedule a function to be called every `interval` seconds, beginning at a time that does not coincide with other scheduled events.

This method is similar to `schedule_interval`, except that the clock will move the interval out of phase with other scheduled functions so as to distribute CPU more load evenly over time.

This is useful for functions that need to be called regularly, but not relative to the initial start time. pyglet.media does this for scheduling audio buffer updates, which need to occur regularly – if all audio updates are scheduled at the same time (for example, mixing several tracks of a music score, or playing multiple videos back simultaneously), the resulting load on the CPU is excessive for those intervals but idle outside. Using the soft interval scheduling, the load is more evenly distributed.

Soft interval scheduling can also be used as an easy way to schedule graphics animations out of phase; for example, multiple flags waving in the wind.

**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.

Clock.*schedule_once* *(func, delay, *args, **kwargs)*

Schedule a function to be called once after *delay* seconds.

The callback function prototype is the same as for *schedule*.

**Parameters**

• **func** *(function)* – The function to call when the timer lapses.

• **delay** *(float)* – The number of seconds to wait before the timer lapses.

Clock.*set_fps_limit* *(fps_limit)*

Set the framerate limit.

The framerate limit applies only when a function is scheduled for every frame. That is, the framerate limit can be exceeded by scheduling a function for a very small period of time.

**Parameters**

• **fps_limit** *(float)* – Maximum frames per second allowed, or None to disable limiting.

**Warning:** Deprecated. Use *pyglet.app.run* and *schedule_interval* instead.

Clock.*tick* *(poll=False)*

Signify that one frame has passed.

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.

Clock.*unschedule* *(func)*

Remove a function from the schedule.

If the function appears in the schedule more than once, all occurrences are removed. If the function was not scheduled, no error is raised.

**Parameters**

• **func** *(function)* – The function to remove from the schedule.

Clock.*update_time* *

Get the elapsed time since the last call to *update_time*.

This updates the clock’s internal measure of time and returns the difference since the last update (or since the clock was created).

**Note:** Since pyglet 1.2

**Return type** float

**Returns** The number of seconds since the last *update_time*, or 0 if this was the first time it was called.
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.

Inherited members

Methods

Clock.sleep(microseconds)

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.

Constructor:

__init__ (font=None, interval=0.25, format=’%(fps).2f’, color=(0.5, 0.5, 0.5, 0.5), clock=None)
Create a ClockDisplay.

All parameters are optional. By default, a large translucent font will be used to display the FPS to two
decimal places.

Parameters

• font (pyglet.font.Font) – The font to format text in.
• interval (float) – The number of seconds between updating the display.
• format (str) – A format string describing the format of the text. This string is modulated
  with the dict {’fps’ : fps}.
• color (4-tuple of float) – The color, including alpha, passed to glColor4f.
• clock (Clock) – The clock which determines the time. If None, the default clock is used.
Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>draw()</code></td>
<td>Method called each frame to render the label.</td>
</tr>
<tr>
<td><code>unschedule()</code></td>
<td>Remove the display from its clock's schedule.</td>
</tr>
<tr>
<td><code>update_text</code>((dt))</td>
<td>Scheduled method to update the label text.</td>
</tr>
</tbody>
</table>

Methods

ClockDisplay.**draw()**

Method called each frame to render the label.

ClockDisplay.**unschedule()**

Remove the display from its clock’s schedule.

ClockDisplay uses Clock.schedule_interval to periodically update its display label. Even if the ClockDisplay is not being used any more, its update method will still be scheduled, which can be a resource drain. Call this method to unschedule the update method and allow the ClockDisplay to be garbage collected.

**Note:** Since pyglet 1.1

ClockDisplay.**update_text**(dt=0)

Scheduled method to update the label text.

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_default()</code></td>
<td>Return the Clock instance that is used by all module-level clock functions.</td>
</tr>
<tr>
<td><code>get_fps()</code></td>
<td>Return the current measured FPS of the default clock.</td>
</tr>
<tr>
<td><code>get_fps_limit()</code></td>
<td>Get the framerate limit for the default clock.</td>
</tr>
<tr>
<td><code>get_sleep_time</code></td>
<td>Get the time until the next item is scheduled on the default clock.</td>
</tr>
<tr>
<td><code>schedule()</code></td>
<td>Schedule ‘func’ to be called every frame on the default clock.</td>
</tr>
<tr>
<td><code>schedule_interval</code></td>
<td>Schedule ‘func’ to be called every ‘interval’ seconds on the default clock.</td>
</tr>
<tr>
<td><code>schedule_interval_soft</code></td>
<td>Schedule ‘func’ to be called every ‘interval’ seconds on the default clock, begining at a time that does not coincide with other scheduled events.</td>
</tr>
<tr>
<td><code>schedule_once</code></td>
<td>Schedule ‘func’ to be called once after ‘delay’ seconds (can be a float) on the default clock.</td>
</tr>
<tr>
<td><code>set_default</code></td>
<td>Set the default clock to use for all module-level functions.</td>
</tr>
<tr>
<td><code>set_fps_limit</code></td>
<td>Set the framerate limit for the default clock.</td>
</tr>
<tr>
<td><code>tick</code></td>
<td>Signify that one frame has passed on the default clock.</td>
</tr>
<tr>
<td><code>unschedule()</code></td>
<td>Remove ‘func’ from the default clock’s schedule.</td>
</tr>
</tbody>
</table>

**get_default** Function Defined in pyglet.clock

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

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 = ‘linux2’`
`str(object='') -> string`
Return a nice string representation of the object. If the argument is a string, the return value is the same object.

Notes

Defined

- pyglet
- time

`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:

```python
def on_resize(width, height):
    # ...
dispatcher.on_resize = on_resize
```

There is also a convenience decorator that reduces typing:

```python
@dispatcher.event
def on_resize(width, height):
    # ...
```
You may prefer to subclass and override the event handlers instead:

```python
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:

```python
dispatcher.push_handlers(on_resize, on_key_press)
```

If your function handlers have different names to the events they handle, use keyword arguments:

```python
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:

```python
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:

```python
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.

```python
pyglet.event.EventDispatcher
```

### EventDispatcher Class

class **EventDispatcher**

Generic event dispatcher interface.

See the module docstring for usage.

**Methods:**

- `dispatch_event(event_type, *args)`
  Dispatch a single event to the attached handlers.

- `event(*args)`
  Function decorator for an event handler.

- `pop_handlers()`
  Pop the top level of event handlers off the stack.

- `push_handlers(*args, **kwargs)`
  Push a level onto the top of the handler stack, then attach zero or more event handlers.

- `register_event_type(name)`
  Register an event type with the dispatcher.

- `remove_handler(name, handler)`
  Remove a single event handler.

- `remove_handlers(*args, **kwargs)`
  Remove event handlers from the event stack.

- `set_handler(name, handler)`
  Attach a single event handler.

- `set_handlers(*args, **kwargs)`
  Attach one or more event handlers to the top level of the handler stack.

### Methods

**EventDispatcher**. `dispatch_event(event_type, *args)`

Dispatch a single event to the attached handlers.

The event is propagated to all handlers from from the top of the stack until one returns \texttt{EVENT\_HANDED}. This method should be used only by \texttt{EventDispatcher} implementors; applications should call the \texttt{dispatch\_events} method.

Since pyglet 1.2, the method returns \texttt{EVENT\_HANDED} if an event handler returned \texttt{EVENT\_HANDED} or \texttt{EVENT\_UNHANDED} if all events returned \texttt{EVENT\_UNHANDED}. If no matching event handlers are in the stack, \texttt{False} is returned.

**Parameters**

- \texttt{event\_type (str)} – Name of the event.
- \texttt{args (sequence)} – Arguments to pass to the event handler.

**Return type** \texttt{bool or None}

**Returns** (Since pyglet 1.2) \texttt{EVENT\_HANDED} if an event handler returned \texttt{EVENT\_HANDED}; \texttt{EVENT\_UNHANDED} if one or more event handlers were invoked but returned only \texttt{EVENT\_UNHANDED}; otherwise \texttt{False}. In pyglet 1.1 and earlier, the return value is always \texttt{None}.
EventDispatcher.event(*args)

Function decorator for an event handler.

Usage:

```python
win = window.Window()

@win.event
def on_resize(self, width, height):
    # ...
```

or:

```python
@win.event('on_resize')
def foo(self, width, height):
    # ...
```

EventDispatcher.pop_handlers()

Pop the top level of event handlers off the stack.

EventDispatcher.push_handlers(*args, **kwargs)

Push a level onto the top of the handler stack, then attach zero or more event handlers.

If keyword arguments are given, they name the event type to attach. Otherwise, a callable’s __name__ attribute will be used. Any other object may also be specified, in which case it will be searched for callables with event names.

```python
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.
```

EventDispatcher.remove_handler(name, handler)

Remove a single event handler.

The given event handler is removed from the first handler stack frame it appears in. The handler must be the exact same callable as passed to set_handler, set_handlers or push_handlers; and the name must match the event type it is bound to.

No error is raised if the event handler is not set.

Parameters

- name (str) – Name of the event type to remove.
- handler (callable) – Event handler to remove.

EventDispatcher.remove_handlers(*args, **kwargs)

Remove event handlers from the event stack.

See push_handlers for the accepted argument types. All handlers are removed from the first stack frame that contains any of the given handlers. No error is raised if any handler does not appear in that frame, or if no stack frame contains any of the given handlers.

If the stack frame is empty after removing the handlers, it is removed from the stack. Note that this interferes with the expected symmetry of push_handlers and pop_handlers.

EventDispatcher.set_handler(name, handler)

Attach a single event handler.

Parameters
• **name** (*str*) – Name of the event type to attach to.

• **handler** (*callable*) – Event handler to attach.

**EventDispatcher**.set_handlers (**args**, **kwargs**)  
Attach one or more event handlers to the top level of the handler stack.  
See *push_handlers* for the accepted argument types.

**Exceptions**

<table>
<thead>
<tr>
<th>EventException</th>
<th>An exception raised when an event handler could not be attached.</th>
</tr>
</thead>
</table>

**EventException**  
Exception defined in *pyglet.event*  

```
exception EventException  
An exception EventException 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.extlibs**  
External dependencies for Pyglet.  
These dependencies are included to publish Pyglet as a fully self-contained package.

**Modules**
This Python module implements support for PNG images (see PNG specification at http://www.w3.org/TR/2003/REC-PNG-20031110/). It reads and writes PNG files with all allowable bit depths (1/2/4/8/16/24/32/48/64 bits per pixel) and colour combinations: greyscale (1/2/4/8/16 bit); RGB, RGBA, LA (greyscale with alpha) with 8/16 bits per channel; colour mapped images (1/2/4/8 bit). Adam7 interlacing is supported for reading and writing. A number of optional chunks can be specified (when writing) and understood (when reading): tRNS, bKGD, gAMA.

For help, type import png; help(png) in your python interpreter.

A good place to start is the Reader and Writer classes.

Requires Python 2.3. Limited support is available for Python 2.2, but not everything works. Best with Python 2.4 and higher. Installation is trivial, but see the README.txt file (with the source distribution) for details.

This file can also be used as a command-line utility to convert Netpbm PNM files to PNG, and the reverse conversion from PNG to PNM. The interface is similar to that of the pnmtopng program from Netpbm. Type python png.py --help at the shell prompt for usage and a list of options.

A note on spelling and terminology  Generally British English spelling is used in the documentation. So that’s “greyscale” and “colour”. This not only matches the author’s native language, it’s also used by the PNG specification.

The major colour models supported by PNG (and hence by PyPNG) are: greyscale, RGB, greyscale–alpha, RGB–alpha. These are sometimes referred to using the abbreviations: L, RGB, LA, RGBA. In this case each letter abbreviates a single channel: L is for Luminance or Luma or Lightness which is the channel used in greyscale images; R, G, B stand for Red, Green, Blue, the components of a colour image; A stands for Alpha, the opacity channel (used for transparency effects, but higher values are more opaque, so it makes sense to call it opacity).

A note on formats  When getting pixel data out of this module (reading) and presenting data to this module (writing) there are a number of ways the data could be represented as a Python value. Generally this module uses one of three formats called “flat row flat pixel”, “boxed row flat pixel”, and “boxed row boxed pixel”. Basically the concern is whether each pixel and each row comes in its own little tuple (box), or not.

Consider an image that is 3 pixels wide by 2 pixels high, and each pixel has RGB components:

Boxed row flat pixel:

```
```

Each row appears as its own list, but the pixels are flattened so that three values for one pixel simply follow the three values for the previous pixel. This is the most common format used, because it provides a good compromise between space and convenience. PyPNG regards itself as at liberty to replace any sequence type with any sufficiently compatible other sequence type; in practice each row is an array (from the array module), and the outer list is sometimes an iterator rather than an explicit list (so that streaming is possible).

Flat row flat pixel:

```
```

The entire image is one single giant sequence of colour values. Generally an array will be used (to save space), not a list.

Boxed row boxed pixel:
Each row appears in its own list, but each pixel also appears in its own tuple. A serious memory burn in Python.

In all cases the top row comes first, and for each row the pixels are ordered from left-to-right. Within a pixel the values appear in the order, R-G-B-A (or L-A for greyscale–alpha).

There is a fourth format, mentioned because it is used internally, is close to what lies inside a PNG file itself, and has some support from the public API. This format is called packed. When packed, each row is a sequence of bytes (integers from 0 to 255), just as it is before PNG scanline filtering is applied. When the bit depth is 8 this is essentially the same as boxed row flat pixel; when the bit depth is less than 8, several pixels are packed into each byte; when the bit depth is 16 (the only value more than 8 that is supported by the PNG image format) each pixel value is decomposed into 2 bytes (and packed is a misnomer). This format is used by the `Writer.write_packed()` method. It isn’t usually a convenient format, but may be just right if the source data for the PNG image comes from something that uses a similar format (for example, 1-bit BMPs, or another PNG file).

And now, my famous members

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>A PNG image.</td>
</tr>
<tr>
<td>Reader</td>
<td>PNG decoder in pure Python.</td>
</tr>
<tr>
<td>Writer</td>
<td>PNG encoder in pure Python.</td>
</tr>
</tbody>
</table>

Classes

**Image Class**

class Image(rows, info)

A PNG image. You can create an Image object from an array of pixels by calling `png.from_array()`. It can be saved to disk with the `save()` method.

**Constructor:**

`__init__`(rows, info)

**Note:** The constructor is not public. Please do not call it.

**Methods:**

`save(file)` Save the image to file.
Methods

Image.save(file)

Save the image to file. If file looks like an open file descriptor then it is used, otherwise it is treated as a filename
and a fresh file is opened.

In general, you can only call this method once; after it has been called the first time and the PNG image has
been saved, the source data will have been streamed, and cannot be streamed again.

Reader Class

class Reader(_guess=None, **kw)

PNG decoder in pure Python.

Constructor:

__init__(_guess=None, **kw)

Create a PNG decoder object.

The constructor expects exactly one keyword argument. If you supply a positional argument instead, it
will guess the input type. You can choose among the following keyword arguments:

filename Name of input file (a PNG file).

guess A file-like object (object with a read() method).

bytes array or string with PNG data.

Methods:

asDirect() Returns the image data as a direct representation of an x * y * planes array.

asFloat([maxval]) Return image pixels as per asDirect() method, but scale all pixel values to be fl

asRGB() Return image as RGB pixels.

asRGB8() Return the image data as an RGB pixels with 8-bits per sample.

asRGBA() Return image as RGBA pixels.

asRGBA8() Return the image data as RGBA pixels with 8-bits per sample.

chunk([seek, lenient]) Read the next PNG chunk from the input file; returns a (type,*data*) tuple.

chunklentype() Reads just enough of the input to determine the next chunk’s length and type, return

chunks() Return an iterator that will yield each chunk as a (chunktype, content) pair.

depinterlace(raw) Read raw pixel data, undo filters, deinterlace, and flatten.

iterboxed(rows) Iterator that yields each scanline in boxed row flat pixel format.

iterstraight(raw) Iterator that undoes the effect of filtering, and yields each row in serialised format (as

palette([alpha]) Returns a palette that is a sequence of 3-tuples or 4-tuples, synthesizing it from the

preamble([lenient]) Extract the image metadata by reading the initial part of the PNG file up to the start

process_chunk([lenient]) Process the next chunk and its data.

read([lenient]) Read the PNG file and decode it.

read_flat() Read a PNG file and decode it into flat row flat pixel format.

serialtoflat(bytes[, width]) Convert serial format (byte stream) pixel data to flat row flat pixel.

undo_filter(filter_type, scanline, previous) Undo the filter for a scanline.

validate_signature() If signature (header) has not been read then read and validate it; otherwise do nothing.
Methods

Reader.asDirect()

- Returns the image data as a direct representation of an \(x \times y \times \text{planes}\) array. This method is intended to remove the need for callers to deal with palettes and transparency themselves. Images with a palette (colour type 3) are converted to RGB or RGBA; images with transparency (a \texttt{tRNS} chunk) are converted to LA or RGBA as appropriate. When returned in this format the pixel values represent the colour value directly without needing to refer to palettes or transparency information.

- Like the \texttt{read()} method this method returns a 4-tuple:
  \[(\text{width}, \text{height}, \text{pixels}, \text{meta})\]

  This method normally returns pixel values with the bit depth they have in the source image, but when the source PNG has an \texttt{sBIT} chunk it is inspected and can reduce the bit depth of the result pixels; pixel values will be reduced according to the bit depth specified in the \texttt{sBIT} chunk (PNG nerds should note a single result bit depth is used for all channels; the maximum of the ones specified in the \texttt{sBIT} chunk. An RGB565 image will be rescaled to 6-bit RGB666).

  The \texttt{meta} dictionary that is returned reflects the \texttt{direct} format and not the original source image. For example, an RGB source image with a \texttt{tRNS} chunk to represent a transparent colour, will have \texttt{planes=3} and \texttt{alpha=False} for the source image, but the \texttt{meta} dictionary returned by this method will have \texttt{planes=4} and \texttt{alpha=True} because an alpha channel is synthesized and added.

- \texttt{pixels} is the pixel data in boxed row flat pixel format (just like the \texttt{read()} method).

- All the other aspects of the image data are not changed.

Reader.asFloat (\texttt{maxval=1.0})

- Return image pixels as per \texttt{asDirect()} method, but scale all pixel values to be floating point values between 0.0 and \texttt{maxval}.

Reader.asRGB()

- Return image as RGB pixels. RGB colour images are passed through unchanged; greyscales are expanded into RGB triplets (there is a small speed overhead for doing this).

  An alpha channel in the source image will raise an exception.

  The return values are as for the \texttt{read()} method except that the \texttt{metadata} reflect the returned pixels, not the source image. In particular, for this method \texttt{metadata[‘greyscale’]} will be \texttt{False}.

Reader.asRGB8()

- Return the image data as an RGB pixels with 8-bits per sample. This is like the \texttt{asRGB()} method except that this method additionally rescales the values so that they are all between 0 and 255 (8-bit). In the case where the source image has a bit depth < 8 the transformation preserves all the information; where the source image has bit depth > 8, then rescaling to 8-bit values loses precision. No dithering is performed. Like \texttt{asRGB()}, an alpha channel in the source image will raise an exception.

  This function returns a 4-tuple: \((\text{width}, \text{height}, \text{pixels}, \text{metadata})\). \texttt{width}, \texttt{height}, \texttt{metadata} are as per the \texttt{read()} method.

  \texttt{pixels} is the pixel data in boxed row flat pixel format.

Reader.asRGBA()

- Return image as RGBA pixels. Greyscales are expanded into RGB triplets; an alpha channel is synthesized if necessary. The return values are as for the \texttt{read()} method except that the \texttt{metadata} reflect the returned pixels, not the source image. In particular, for this method \texttt{metadata[‘greyscale’]} will be \texttt{False}, and \texttt{metadata[‘alpha’]} will be \texttt{True}.
Reader.asRGBA8()

Return the image data as RGBA pixels with 8-bits per sample. This method is similar to asRGB8() and asRgba(): The result pixels have an alpha channel, and values are rescaled to the range 0 to 255. The alpha channel is synthesized if necessary (with a small speed penalty).

Reader.chunk(seek=None, lenient=False)

Read the next PNG chunk from the input file; returns a (type,*data*) tuple. type is the chunk’s type as a string (all PNG chunk types are 4 characters long). data is the chunk’s data content, as a string.

If the optional seek argument is specified then it will keep reading chunks until it either runs out of file or finds the type specified by the argument. Note that in general the order of chunks in PNGs is unspecified, so using seek can cause you to miss chunks.

If the optional lenient argument evaluates to True, checksum failures will raise warnings rather than exceptions.

Reader.chunklentype()

Reads just enough of the input to determine the next chunk’s length and type, returned as a (length, type) pair where type is a string. If there are no more chunks, None is returned.

Reader.chunks()

Return an iterator that will yield each chunk as a (chunktype, content) pair.

Reader.deinterlace(raw)

Read raw pixel data, undo filters, deinterlace, and flatten. Return in flat row flat pixel format.

Reader.iterboxed(rows)

Iterator that yields each scanline in boxed row flat pixel format. rows should be an iterator that yields the bytes of each row in turn.

Reader.iterstraight(raw)

Iterator that undoes the effect of filtering, and yields each row in serialised format (as a sequence of bytes). Assumes input is straightlaced. raw should be an iterable that yields the raw bytes in chunks of arbitrary size.

Reader.palette(alpha='natural')

Returns a palette that is a sequence of 3-tuples or 4-tuples, synthesizing it from the PLTE and tRNS chunks. These chunks should have already been processed (for example, by calling the preamble() method). All the tuples are the same size: 3-tuples if there is no tRNS chunk, 4-tuples when there is a tRNS chunk. Assumes that the image is colour type 3 and therefore a PLTE chunk is required.

If the alpha argument is ‘force’ then an alpha channel is always added, forcing the result to be a sequence of 4-tuples.

Reader.preamble(lenient=False)

Extract the image metadata by reading the initial part of the PNG file up to the start of the IDAT chunk. All the chunks that precede the IDAT chunk are read and either processed for metadata or discarded.

If the optional lenient argument evaluates to True, checksum failures will raise warnings rather than exceptions.

Reader.process_chunk(lenient=False)

Process the next chunk and its data. This only processes the following chunk types, all others are ignored: IHDR, PLTE, bKGD, tRNS, gAMA, sBIT.

If the optional lenient argument evaluates to True, checksum failures will raise warnings rather than exceptions.

Reader.read(lenient=False)

Read the PNG file and decode it. Returns (width, height, pixels, metadata).

May use excessive memory.

pixels are returned in boxed row flat pixel format.

If the optional lenient argument evaluates to True, checksum failures will raise warnings rather than exceptions.
Reader.\texttt{read\_flat}()
Read a PNG file and decode it into flat row flat pixel format. Returns \((width, height, pixels, metadata)\).

May use excessive memory.

\textit{pixels} are returned in flat row flat pixel format.

See also the \texttt{read()} method which returns pixels in the more stream-friendly boxed row flat pixel format.

Reader.\texttt{serialtoflat}(\textit{bytes}, width=None)
Convert serial format (byte stream) pixel data to flat row flat pixel.

Reader.\texttt{undo\_filter}(\textit{filter\_type}, \textit{scanline}, previous)
Undo the filter for a scanline. \textit{scanline} is a sequence of bytes that does not include the initial filter type byte. \textit{previous} is decoded previous scanline (for straightlaced images this is the previous pixel row, but for interlaced images, it is the previous scanline in the reduced image, which in general is not the previous pixel row in the final image). When there is no previous scanline (the first row of a straightlaced image, or the first row in one of the passes in an interlaced image), then this argument should be \texttt{None}.

The scanline will have the effects of filtering removed, and the result will be returned as a fresh sequence of bytes.

Reader.\texttt{validate\_signature}()
If signature (header) has not been read then read and validate it; otherwise do nothing.

\begin{center}
\textbf{pyglet.extlibs.png.Writer}
\end{center}

\textbf{Writer Class}

\texttt{class Writer}(width=None, height=None, size=None, greyscale=False, alpha=False, bitdepth=8, palette=None, transparent=None, background=None, gamma=None, compression=None, interlace=False, bytes_per_sample=None, planes=None, colormap=None, maxval=None, chunk_limit=1048576)
PNG encoder in pure Python.

\textbf{Constructor:}

\texttt{\_\_init\_}(width=None, height=None, size=None, greyscale=False, alpha=False, bitdepth=8, palette=None, transparent=None, background=None, gamma=None, compression=None, interlace=False, bytes_per_sample=None, planes=None, colormap=None, maxval=None, chunk_limit=1048576)
Create a PNG encoder object.

Arguments:

\textbf{width, height} Image size in pixels, as two separate arguments.
\textbf{size} Image size (w,h) in pixels, as single argument.
\textbf{greyscale} Input data is greyscale, not RGB.
\textbf{alpha} Input data has alpha channel (RGBA or LA).
\textbf{bitdepth} Bit depth: from 1 to 16.
\textbf{palette} Create a palette for a colour mapped image (colour type 3).
transparent Specify a transparent colour (create a tRNS chunk).

background Specify a default background colour (create a bKGD chunk).

gamma Specify a gamma value (create a gAMA chunk).

compression zlib compression level: 0 (none) to 9 (more compressed); default: -1 or None.

interlace Create an interlaced image.

chunk_limit Write multiple IDAT chunks to save memory.

The image size (in pixels) can be specified either by using the width and height arguments, or with the single size argument. If size is used it should be a pair (width, height).

greyscale and alpha are booleans that specify whether an image is greyscale (or colour), and whether it has an alpha channel (or not).

bitdepth specifies the bit depth of the source pixel values. Each source pixel value must be an integer between 0 and \(2^{\text{bitdepth}} - 1\). For example, 8-bit images have values between 0 and 255. PNG only stores images with bit depths of 1,2,4,8, or 16. When bitdepth is not one of these values, the next highest valid bit depth is selected, and an sBIT (significant bits) chunk is generated that specifies the original precision of the source image. In this case the supplied pixel values will be rescaled to fit the range of the selected bit depth.

The details of which bit depth / colour model combinations the PNG file format supports directly, are somewhat arcane (refer to the PNG specification for full details). Briefly: “small” bit depths (1,2,4) are only allowed with greyscale and colour mapped images; colour mapped images cannot have bit depth 16.

For colour mapped images (in other words, when the palette argument is specified) the bitdepth argument must match one of the valid PNG bit depths: 1, 2, 4, or 8. (It is valid to have a PNG image with a palette and an sBIT chunk, but the meaning is slightly different; it would be awkward to press the bitdepth argument into service for this.)

The palette option, when specified, causes a colour mapped image to be created: the PNG colour type is set to 3; greyscale must not be set; alpha must not be set; transparent must not be set; the bit depth must be 1,2,4, or 8. When a colour mapped image is created, the pixel values are palette indexes and the bitdepth argument specifies the size of these indexes (not the size of the colour values in the palette).

The palette argument value should be a sequence of 3- or 4-tuples. 3-tuples specify RGB palette entries; 4-tuples specify RGBA palette entries. If both 4-tuples and 3-tuples appear in the sequence then all the 4-tuples must come before all the 3-tuples. A PLTE chunk is created; if there are 4-tuples then a tRNS chunk is created as well. The PLTE chunk will contain all the RGB triples in the same sequence; the tRNS chunk will contain the alpha channel for all the 4-tuples, in the same sequence. Palette entries are always 8-bit.

If specified, the transparent and background parameters must be a tuple with three integer values for red, green, blue, or a simple integer (or singleton tuple) for a greyscale image.

If specified, the gamma parameter must be a positive number (generally, a float). A gAMA chunk will be created. Note that this will not change the values of the pixels as they appear in the PNG file, they are assumed to have already been converted appropriately for the gamma specified.

The compression argument specifies the compression level to be used by the zlib module. Values from 1 to 9 specify compression, with 9 being “more compressed” (usually smaller and slower, but it doesn’t always work out that way). 0 means no compression. -1 and None both mean that the default level of compression will be picked by the zlib module (which is generally acceptable).

If interlace is true then an interlaced image is created (using PNG’s so far only interlace method, Adam7). This does not affect how the pixels should be presented to the encoder, rather it changes how they are arranged into the PNG file. On slow connections interlaced images can be partially decoded by the browser to give a rough view of the image that is successively refined as more image data appears.
Note: Enabling the *interlace* option requires the entire image to be processed in working memory.

Chunk limit is used to limit the amount of memory used whilst compressing the image. In order to avoid using large amounts of memory, multiple IDAT chunks may be created.

### Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array_scanlines(pixels)</code></td>
<td>Generates boxed rows (flat pixels) from flat rows (flat pixels) in an array.</td>
</tr>
<tr>
<td><code>array_scanlines_interlace(pixels)</code></td>
<td>Generator for interlaced scanlines from an array.</td>
</tr>
<tr>
<td><code>convert_pnm(infile, outfile)</code></td>
<td>Convert a PNM file containing raw pixel data into a PNG file with the parameters set in the writer object. Works for (binary) PGM, PPM, and PAM formats.</td>
</tr>
<tr>
<td><code>convert_ppm_and_pgm(ppmfile, pgmfile, outfile)</code></td>
<td>Convert a PPM and PGM file containing raw pixel data into a PNG outfile with the parameters set in the writer object.</td>
</tr>
<tr>
<td><code>file_scanlines(infile)</code></td>
<td>Generates boxed rows in flat pixel format, from the input file <code>infile</code>.</td>
</tr>
<tr>
<td><code>make_palette()</code></td>
<td>Create the byte sequences for a PLTE and if necessary a tRNS chunk.</td>
</tr>
<tr>
<td><code>write(outfile, rows)</code></td>
<td>Write a PNG image to the output file.</td>
</tr>
<tr>
<td><code>write_array(outfile, pixels)</code></td>
<td>Write an array in flat row flat pixel format as a PNG file on the output file.</td>
</tr>
<tr>
<td><code>write_packed(outfile, rows)</code></td>
<td>Write PNG file to <code>outfile</code>.</td>
</tr>
<tr>
<td><code>write_passes(outfile, rows[, packed])</code></td>
<td>Write a PNG image to the output file.</td>
</tr>
</tbody>
</table>

### Note:

Interlacing will require the entire image to be in working memory.
**pyglet Documentation, Release 1.2.4**

**Writer.**\texttt{write\_array} (\texttt{outfile}, \texttt{pixels})

Write an array in flat row flat pixel format as a PNG file on the output file. See also \texttt{write()} method.

**Writer.**\texttt{write\_packed} (\texttt{outfile}, \texttt{rows})

Write PNG file to \texttt{outfile}. The pixel data comes from \texttt{rows} which should be in boxed row packed format. Each row should be a sequence of packed bytes.

Technically, this method does work for interlaced images but it is best avoided. For interlaced images, the rows should be presented in the order that they appear in the file.

This method should not be used when the source image bit depth is not one naturally supported by PNG; the bit depth should be 1, 2, 4, 8, or 16.

**Writer.**\texttt{write\_passes} (\texttt{outfile}, \texttt{rows}, \texttt{packed=False})

Write a PNG image to the output file.

Most users are expected to find the \texttt{write()} or \texttt{write\_array()} method more convenient.

The \texttt{rows} should be given to this method in the order that they appear in the output file. For straightlaced images, this is the usual top to bottom ordering, but for interlaced images the rows should have already been interlaced before passing them to this function.

\texttt{rows} should be an iterable that yields each row. When \texttt{packed} is \texttt{False} the rows should be in boxed row flat pixel format; when \texttt{packed} is \texttt{True} each row should be a packed sequence of bytes.

**pngfilters** Class

```python
class pngfilters
```

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChunkError</td>
<td>Exception defined in \texttt{pyglet.extlibs.png}</td>
</tr>
</tbody>
</table>

**Exceptions**

```python
pyglet.extlibs.png.Error → pyglet.extlibs.pngFormatException → pyglet.extlibs.png.ChunkError
```
Error
Exception defined in `pyglet.extlibs.png`

```python
pyglet.extlibs.png.Error
```

Exception `Error`
Problem with input file format. In other words, PNG file does not conform to the specification in some way and is invalid.

```python
pyglet.extlibs.png.Error → pyglet.extlibs.png.FormatError
```

FormatException
Exception defined in `pyglet.extlibs.png`

```python
pyglet.extlibs.png.Error → pyglet.extlibs.png.FormatError
```

Problem with input file format. In other words, PNG file does not conform to the specification in some way and is invalid.

### Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>check_bitdepth_colortype</code></td>
<td>Check that <code>bitdepth</code> and <code>colortype</code> are both valid, and specified in a valid combination.</td>
</tr>
<tr>
<td><code>check_color</code></td>
<td>Checks that a colour argument for transparent or background options is the right form.</td>
</tr>
<tr>
<td><code>checkPalette</code></td>
<td>Check a palette argument (to the <code>Writer</code> class) for validity.</td>
</tr>
<tr>
<td><code>check_sizes</code></td>
<td>Check that these arguments, in supplied, are consistent.</td>
</tr>
<tr>
<td><code>color_triple</code></td>
<td>Convert a command line colour value to a RGB triple of integers.</td>
</tr>
<tr>
<td><code>filter_scanline</code></td>
<td>Apply a scanline filter to a scanline.</td>
</tr>
<tr>
<td><code>from_array</code></td>
<td>Create a PNG <code>Image</code> object from a 2- or 3-dimensional array.</td>
</tr>
<tr>
<td><code>fromarray</code></td>
<td>Create a PNG <code>Image</code> object from a 2- or 3-dimensional array.</td>
</tr>
<tr>
<td><code>group</code></td>
<td></td>
</tr>
<tr>
<td><code>interleave_planes</code></td>
<td>Interleave (colour) planes, e.g.</td>
</tr>
<tr>
<td><code>isarray</code></td>
<td>Same as <code>isinstance(x, array)</code> except on Python 2.2, where it always returns False.</td>
</tr>
<tr>
<td><code>isinteger</code></td>
<td></td>
</tr>
<tr>
<td><code>read_pam_header</code></td>
<td>Read (the rest of a) PAM header.</td>
</tr>
<tr>
<td><code>read_pnm_header</code></td>
<td>Read a PNM header, returning (format,width,height,depth,maxval).</td>
</tr>
<tr>
<td><code>tostring</code></td>
<td>Convert row of bytes to string.</td>
</tr>
<tr>
<td><code>write_chunk</code></td>
<td>Write a PNG chunk to the output file, including length and checksum.</td>
</tr>
<tr>
<td><code>write_chunks</code></td>
<td>Create a PNG file by writing out the chunks.</td>
</tr>
<tr>
<td><code>write_pnm</code></td>
<td>Write a Netpbm PNM/PAM file.</td>
</tr>
</tbody>
</table>

### Functions

- **`check_bitdepth_colortype` Function** Defined in `pyglet.extlibs.png`

```python
check_bitdepth_colortype(bitdepth, colortype)
```

2.1. **pyglet** 115
Check that `bitdepth` and `colortype` are both valid, and specified in a valid combination. Returns if valid, raise an Exception if not valid.

**check_color Function**  Defined in `pyglet.extlibs.png`

**check_color** *(c, greyscale, which)*

Checks that a colour argument for transparent or background options is the right form. Returns the colour (which, if it’s a bar integer, is “corrected” to a 1-tuple).

**check_palette Function**  Defined in `pyglet.extlibs.png`

**check_palette** *(palette)*

Check a palette argument (to the Writer class) for validity. Returns the palette as a list if okay; raises an exception otherwise.

**check_sizes Function**  Defined in `pyglet.extlibs.png`

**check_sizes** *(size, width, height)*

Check that these arguments, in supplied, are consistent. Return a (width, height) pair.

**color_triple Function**  Defined in `pyglet.extlibs.png`

**color_triple** *(color)*

Convert a command line colour value to a RGB triple of integers. FIXME: Somewhere we need support for greyscale backgrounds etc.

**filter_scanline Function**  Defined in `pyglet.extlibs.png`

**filter_scanline** *(type, line, fo, prev=None)*

Apply a scanline filter to a scanline. `type` specifies the filter type (0 to 4); `line` specifies the current (unfiltered) scanline as a sequence of bytes; `prev` specifies the previous (unfiltered) scanline as a sequence of bytes. `fo` specifies the filter offset; normally this is size of a pixel in bytes (the number of bytes per sample times the number of channels), but when this is < 1 (for bit depths < 8) then the filter offset is 1.

**from_array Function**  Defined in `pyglet.extlibs.png`

**from_array** *(a, mode=None, info={})*

Create a PNG Image object from a 2- or 3-dimensional array. One application of this function is easy PIL-style saving: `png.from_array(pixels, ’L’).save(‘foo.png’)`.

Unless they are specified using the `info` parameter, the PNG’s height and width are taken from the array size. For a 3 dimensional array the first axis is the height; the second axis is the width; and the third axis is the channel number. Thus an RGB image that is 16 pixels high and 8 wide will use an array that is 16x8x3. For 2 dimensional arrays the first axis is the height, but the second axis is width*channels, so an RGB image that is 16 pixels high and 8 wide will use a 2-dimensional array that is 16x24 (each row will be 8*3==24 sample values).

`mode` is a string that specifies the image colour format in a PIL-style mode. It can be:

- `'L'` greyscale (1 channel)
- `'LA'` greyscale with alpha (2 channel)
- `'RGB'` colour image (3 channel)
- `'RGBA'` colour image with alpha (4 channel)
The mode string can also specify the bit depth (overriding how this function normally derives the bit depth, see below). Appending ';16' to the mode will cause the PNG to be 16 bits per channel; any decimal from 1 to 16 can be used to specify the bit depth.

When a 2-dimensional array is used mode determines how many channels the image has, and so allows the width to be derived from the second array dimension.

The array is expected to be a numpy array, but it can be any suitable Python sequence. For example, a list of lists can be used: `png.from_array([[0, 255, 0], [255, 0, 255]], 'L')`. The exact rules are: `len(a)` gives the first dimension, height; `len(a[0])` gives the second dimension; `len(a[0][0])` gives the third dimension, unless an exception is raised in which case a 2-dimensional array is assumed. It’s slightly more complicated than that because an iterator of rows can be used, and it all still works. Using an iterator allows data to be streamed efficiently.

The bit depth of the PNG is normally taken from the array element’s datatype (but if mode specifies a bitdepth then that is used instead). The array element’s datatype is determined in a way which is supposed to work both for numpy arrays and for Python array.array objects. A 1 byte datatype will give a bit depth of 8, a 2 byte datatype will give a bit depth of 16. If the datatype does not have an implicit size, for example it is a plain Python list of lists, as above, then a default of 8 is used.

The info parameter is a dictionary that can be used to specify metadata (in the same style as the arguments to the `png.Writer` class). For this function the keys that are useful are:

- `height` overrides the height derived from the array dimensions and allows a to be an iterable.
- `width` overrides the width derived from the array dimensions.
- `bitdepth` overrides the bit depth derived from the element datatype (but must match mode if that also specifies a bit depth).

Generally anything specified in the info dictionary will override any implicit choices that this function would otherwise make, but must match any explicit ones. For example, if the info dictionary has a `greyscale` key then this must be true when mode is 'L' or 'LA' and false when mode is 'RGB' or 'RGBA'.

**fromarray Function** Defined in `pyglet.extlibs.png`

`fromarray(a, mode=None, info={})`

Create a PNG Image object from a 2- or 3-dimensional array. One application of this function is easy PIL-style saving: `png.from_array(pixels, 'L').save('foo.png')`.

Unless they are specified using the info parameter, the PNG’s height and width are taken from the array size. For a 3 dimensional array the first axis is the height; the second axis is the width; and the third axis is the channel number. Thus an RGB image that is 16 pixels high and 8 wide will use an array that is 16x8x3. For 2 dimensional arrays the first axis is the height, but the second axis is width*channels, so an RGB image that is 16 pixels high and 8 wide will use a 2-dimensional array that is 16x24 (each row will be 8*3==24 sample values).

`mode` is a string that specifies the image colour format in a PIL-style mode. It can be:

- 'L' greyscale (1 channel)
- 'LA' greyscale with alpha (2 channel)
- 'RGB' colour image (3 channel)
- 'RGBA' colour image with alpha (4 channel)

The mode string can also specify the bit depth (overriding how this function normally derives the bit depth, see below). Appending ';16' to the mode will cause the PNG to be 16 bits per channel; any decimal from 1 to 16 can be used to specify the bit depth.
When a 2-dimensional array is used mode determines how many channels the image has, and so allows the width to be derived from the second array dimension.

The array is expected to be a numpy array, but it can be any suitable Python sequence. For example, a list of lists can be used: png.from_array([[0, 255, 0], [255, 0, 255]], 'L'). The exact rules are: len(a) gives the first dimension, height; len(a[0]) gives the second dimension; len(a[0][0]) gives the third dimension, unless an exception is raised in which case a 2-dimensional array is assumed. It's slightly more complicated than that because an iterator of rows can be used, and it all still works. Using an iterator allows data to be streamed efficiently.

The bit depth of the PNG is normally taken from the array element’s datatype (but if mode specifies a bitdepth then that is used instead). The array element’s datatype is determined in a way which is supposed to work both for numpy arrays and for Python array.array objects. A 1 byte datatype will give a bit depth of 8, a 2 byte datatype will give a bit depth of 16. If the datatype does not have an implicit size, for example it is a plain Python list of lists, as above, then a default of 8 is used.

The info parameter is a dictionary that can be used to specify metadata (in the same style as the arguments to the :class:png.Writer class). For this function the keys that are useful are:

- **height** overrides the height derived from the array dimensions and allows a to be an iterable.
- **width** overrides the width derived from the array dimensions.
- **bitdepth** overrides the bit depth derived from the element datatype (but must match mode if that also specifies a bit depth).

Generally anything specified in the info dictionary will override any implicit choices that this function would otherwise make, but must match any explicit ones. For example, if the info dictionary has a greyscale key then this must be true when mode is 'L' or 'LA' and false when mode is 'RGB' or 'RGBA'.

**group Function** Defined in pyglet.extlibs.png

**group(s, n)**

**interleave_planes Function** Defined in pyglet.extlibs.png

**interleave_planes(ipixels, apixels, ipsize, apsize)**

Interleave (colour) planes, e.g. RGB + A = RGBA.

Return an array of pixels consisting of the ipsize elements of data from each pixel in ipixels followed by the apsize elements of data from each pixel in apixels. Conventionally ipixels and apixels are byte arrays so the sizes are bytes, but it actually works with any arrays of the same type. The returned array is the same type as the input arrays which should be the same type as each other.

**isarray Function** Defined in pyglet.extlibs.png

**isarray(x)**

Same as isinstance(x, array) except on Python 2.2, where it always returns False. This helps PyPNG work on Python 2.2.

**isinteger Function** Defined in pyglet.extlibs.png

**isinteger(x)**
**read_pam_header Function**  Defined in pyglet.extlibs.png

**read_pam_header (infile)**
Read (the rest of a) PAM header.  *infile* should be positioned immediately after the initial ‘P7’ line (at the beginning of the second line). Returns are as for read_pnm_header.

**read_pnm_header Function**  Defined in pyglet.extlibs.png

**read_pnm_header (infile, supported=('P5', 'P6'))**
Read a PNM header, returning (format,width,height,depth,maxval).  *width* and *height* are in pixels.  *depth* is the number of channels in the image; for PBM and PGM it is synthesized as 1, for PPM as 3; for PAM images it is read from the header.  *maxval* is synthesized (as 1) for PBM images.

**tostring Function**  Defined in pyglet.extlibs.png

**tostring (row)**
Convert row of bytes to string.  Expects *row* to be an array.

**write_chunk Function**  Defined in pyglet.extlibs.png

**write_chunk (outfile, tag, data='')**
Write a PNG chunk to the output file, including length and checksum.

**write_chunks Function**  Defined in pyglet.extlibs.png

**write_chunks (out, chunks)**
Create a PNG file by writing out the chunks.

**write_pnm Function**  Defined in pyglet.extlibs.png

**write_pnm (file, width, height, pixels, meta)**
Write a Netpbm PNM/PAM file.

**Variables**

```py
generators = _Feature((2, 2, 0, 'alpha', 1), (2, 3, 0, 'final', 0), 0)
```

---

**Defined**

- itertools
- math
- operator
- struct
- sys
- warnings
- zlib

**Notes**
**pyglet.font**

Load fonts and render text.

This is a fairly-low level interface to text rendering. Obtain a font using `load`:

```python
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:

```python
text = 'Hello, world!'
glyphs = arial.get_glyphs(text)
```

The most efficient way to render these glyphs is with a *GlyphString*:

```python
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*:

```python
text = Text(font, text)
text.draw()
```

See the *pyglet.font.base* module for documentation on the base classes used by this package.

---

**Modules**

<table>
<thead>
<tr>
<th>fontconfig</th>
<th>Wrapper around the Linux FontConfig library.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttf</td>
<td>Implementation of the Truetype file format.</td>
</tr>
</tbody>
</table>

**pyglet.font.fontconfig**  Wrapper around the Linux FontConfig library. Used to find available fonts.

---

| FcValue | |
| FontConfig | |
| FontConfigPattern | |
| FontConfigSearchPattern | |
| FontConfigSearchResult | |

**Classes**
**FcValue Class**

class FcValue

Attributes:

- `type` Type: CField
- `u` Type: CField

Attributes

FcValue.type
- Structure/Union member

FcValue.u
- Structure/Union member

**FontConfig Class**

class FontConfig

Constructor:

- `__init__()`

Methods:

- `char_index(face, character)`
- `create_search_pattern()`
- `dispose()`
- `find_font(name[, size, bold, italic])`

Methods

FontConfig.char_index(face, character)
FontConfig.create_search_pattern()
FontConfig.dispose()
FontConfig.find_font(name[, size=12, bold=False, italic=False]
FontConfigPattern Class

class FontConfigPattern (fontconfig, pattern=None)

Constructor:

__init__ (fontconfig, pattern=None)

Attributes:

is_valid

Attributes

FontConfigPattern.is_valid

FontConfigSearchPattern Class

class FontConfigSearchPattern (fontconfig)

Constructor:

__init__ (fontconfig)

Methods:

match()

Attributes:

is_valid

Methods

FontConfigSearchPattern.match()

Inherited members
Attributes

FontConfigSearchPattern.is_valid

FontConfigSearchResult Class

class FontConfigSearchResult (fontconfig, result_pattern)

Constructor:

__init__ (fontconfig, result_pattern)

Methods:

dispose()

Attributes:

bold
face
file
is_valid
italic
name
size

Methods

FontConfigSearchResult.dispose()
Attributes

`FontConfigSearchResult.is_valid`

### Functions

**get_fontconfig**

Defined in `pyglet.font.fontconfig`

```python
get_fontconfig()
```

### Variables

- **FC_FAMILY** = `'family'`
  ```python
  str(object='') -> string
  Return a nice string representation of the object. If the argument is a string, the return value is the same object.
  ```

- **FC_FILE** = `'file'`
  ```python
  str(object='') -> string
  Return a nice string representation of the object. If the argument is a string, the return value is the same object.
  ```

- **FC_FT_FACE** = `'ftface'`
  ```python
  str(object='') -> string
  Return a nice string representation of the object. If the argument is a string, the return value is the same object.
  ```

- **FC_SIZE** = `'size'`
  ```python
  str(object='') -> string
  Return a nice string representation of the object. If the argument is a string, the return value is the same object.
  ```

- **FC_SLANT** = `'slant'`
  ```python
  str(object='') -> string
  Return a nice string representation of the object. If the argument is a string, the return value is the same object.
  ```

- **FC_SLANT_ITALIC** = `100`
  ```python
  int(x=0) -> int or long int(x, base=10) -> int or long
  Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

  If x is not a number or if base is given, then x must be a string or Unicode object 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. >>>

  ```
  \[
  \text{int('0b100', base=0) = 4}
  \]

- **FC_SLANT_ROMAN** = `0`
  ```python
  int(x=0) -> int or long int(x, base=10) -> int or long
  Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

  If x is not a number or if base is given, then x must be a string or Unicode object 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. >>>

  ```
  \[
  \text{int('0b100', base=0) = 4}
  \]
FC_WEIGHT = 'weight'
str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

FC_WEIGHT_BOLD = 200
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FC_WEIGHT_REGULAR = 80
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcMatchFont = 1
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcMatchPattern = 0
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcResultMatch = 0
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcResultNoId = 3
int(x=0) -> int or long int(x, base=10) -> int or long
Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcResultNoMatch = 1
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcResultOutOfMemory = 4
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcResultTypeMismatch = 2
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcTypeBool = 4
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcType CharSet = 6
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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
FcTypeDouble = 2

int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcTypeFTFace = 7

int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcTypeInteger = 1

int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcTypeLangSet = 8

int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcTypeMatrix = 5

int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

FcTypeString = 3

int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.
If x is not a number or if base is given, then x must be a string or Unicode object 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

\texttt{FCTypeVoid} = 0
int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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

pyglet Documentation, Release 1.2.4

pyglet.font.ttf
Implementation of the TrueType file format.
Typical applications will not need to use this module directly; look at pyglet.font instead.

References:
- http://www.microsoft.com/typography/otspec

TruetypeInfo
Information about a single TrueType face.

Classes

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.

Constructor:
__init__ (filename)
Read the given TrueType file.

**Parameters** filename – The name of any Windows, OS2 or Macintosh TrueType file.

The object must be closed (see close) after use.

An exception will be raised if the file does not exist or cannot be read.

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>close()</td>
<td>Close the font file.</td>
</tr>
<tr>
<td>get_character_advances()</td>
<td>Return a dictionary of character-&gt;advance.</td>
</tr>
<tr>
<td>get_character_kernings()</td>
<td>Return a dictionary of (left,right)-&gt;kerning</td>
</tr>
<tr>
<td>get_character_map()</td>
<td>Return the character map.</td>
</tr>
<tr>
<td>get_font_selection_flags()</td>
<td>Return the font selection flags, as defined in OS/2 table</td>
</tr>
<tr>
<td>get_glyph_advances()</td>
<td>Return a dictionary of glyph-&gt;advance.</td>
</tr>
<tr>
<td>get_glyph_kernings()</td>
<td>Return a dictionary of (left,right)-&gt;kerning</td>
</tr>
<tr>
<td>get_glyph_map()</td>
<td>Calculate and return a reverse character map.</td>
</tr>
<tr>
<td>get_horizontal_metrics()</td>
<td>Return all horizontal metric entries in table format.</td>
</tr>
<tr>
<td>get_name(name[, platform, languages])</td>
<td>Returns the value of the given name in this font.</td>
</tr>
<tr>
<td>get_names()</td>
<td>Returns a dictionary of names defined in the file.</td>
</tr>
<tr>
<td>is_bold()</td>
<td>Returns True iff the font describes itself as bold.</td>
</tr>
<tr>
<td>is_italic()</td>
<td>Returns True iff the font describes itself as italic.</td>
</tr>
</tbody>
</table>

**Methods**

*TruetypeInfo.close()*

Close the font file.

This is a good idea, since the entire file is memory mapped in until this method is called. After closing cannot rely on the get_* methods.

*TruetypeInfo.get_character_advances()*

Return a dictionary of character->advance.

They key of the dictionary is a unit-length unicode string, and the value is a float giving the horizontal advance in em.

*TruetypeInfo.get_character_kernings()*

Return a dictionary of (left,right)->kerning

The key of the dictionary is a tuple of (left, right) where each element is a unit-length unicode string. The value of the dictionary is the horizontal pairwise kerning in em.

*TruetypeInfo.get_character_map()*

Return the character map.

Returns a dictionary where the key is a unit-length unicode string and the value is a glyph index. Currently only format 4 character maps are read.

*TruetypeInfo.get_font_selection_flags()*

Return the font selection flags, as defined in OS/2 table

*TruetypeInfo.get_glyph_advances()*

Return a dictionary of glyph->advance.

They key of the dictionary is the glyph index and the value is a float giving the horizontal advance in em.
TruetypeInfo.get_glyph_kernings()
Return a dictionary of (left, right)->kerning

The key of the dictionary is a tuple of (left, right) where each element is a glyph index. The value of
the dictionary is the horizontal pairwise kerning in em.

TruetypeInfo.get_glyph_map()
Calculate and return a reverse character map.

Returns a dictionary where the key is a glyph index and the value is a unit-length unicode string.

TruetypeInfo.get_horizontal_metrics()
Return all horizontal metric entries in table format.

TruetypeInfo.get_name (name, platform=None, languages=None)
Returns the value of the given name in this font.

Parameters

• name – Either an integer, representing the name_id desired (see font format); or a string
describing it, see below for valid names.
• platform – Platform for the requested name. Can be the integer ID, or a string describing
it. By default, the Microsoft platform is searched first, then Macintosh.
• languages – A list of language IDs to search. The first language which defines the re-
quested name will be used. By default, all English dialects are searched.

If the name is not found, None is returned. If the name is found, the value will be decoded and returned as a
unicode string. Currently only some common encodings are supported.

Valid names to request are (supply as a string):
'
copyright'
'family'
'subfamily'
'identifier'
'name'
'version'
'postscript'
'trademark'
'manufacturer'
'designer'
'description'
'vendor-url'
'designer-url'
'license'
'license-url'
'preferred-family'
'preferred-subfamily'
'compatible-name'
'sample'

Valid platforms to request are (supply as a string):
'
'unicode'
'macintosh'
'iso'
'microsoft'
'custom'

TruetypeInfo.get_names()
Returns a dictionary of names defined in the file.
The key of each item is a tuple of `platform_id, name_id`, where each ID is the number as described in the TrueType format.

The value of each item is a tuple of `encoding_id, language_id, value`, where `value` is an encoded string.

`TruetypeInfo.is_bold()`
Returns True iff the font describes itself as bold.

`TruetypeInfo.is_italic()`
Returns True iff the font describes itself as italic.

### Defined
- codecs
- mmap
- os
- struct

### Notes

### Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GlyphString</td>
<td>An immutable string of glyphs that can be rendered quickly.</td>
</tr>
<tr>
<td>Text</td>
<td>Simple displayable text.</td>
</tr>
</tbody>
</table>

```python
pyglet.font.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, the repeatedly call `draw` for each breakpoint.

**Warning:** Deprecated. Use `pyglet.text.layout` classes.

**Constructor:**

```python
__init__ (text, glyphs, x=0, y=0)
```
Create a glyph string.

The `text` string is used to determine valid breakpoints; all glyphs must have already been determined using `pyglet.font.base.Font.get_glyphs`. The string will be positioned with the baseline of the left-most glyph at the given coordinates.

**Parameters**
• **text** (*str or unicode*) – String to represent.
• **glyphs** (list of `pyglet.font.base.Glyph`) – Glyphs representing `text`.
• **x** (*float*) – X coordinate of the left-side bearing of the left-most glyph.
• **y** (*float*) – Y coordinate of the baseline.

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>draw([from_index, to_index])</code></td>
<td>Draw a region of the glyph string.</td>
</tr>
<tr>
<td><code>get_break_index(from_index, width)</code></td>
<td>Find a breakpoint within the text for a given width.</td>
</tr>
<tr>
<td><code>get_subwidth(from_index, to_index)</code></td>
<td>Return the width of a slice of this string.</td>
</tr>
</tbody>
</table>

**Methods**

GlyphString. **draw** *(from_index=0, to_index=None)*

Draw a region of the glyph string.

Assumes texture state is enabled. To enable the texture state:

```python
from pyglet.gl import *
glEnable(GL_TEXTURE_2D)
```

**Parameters**

• **from_index** (*int*) – Start index of text to render.
• **to_index** (*int*) – End index (exclusive) of text to render.

GlyphString. **get_break_index** *(from_index, width)*

Find a breakpoint within the text for a given width.

Returns a valid breakpoint after `from_index` so that the text between `from_index` and the breakpoint fits within `width` pixels.

This method uses precomputed cumulative glyph widths to give quick answer, and so is much faster than `pyglet.font.base.Font.get_glyphs_for_width`.

**Parameters**

• **from_index** (*int*) – Index of text to begin at, or 0 for the beginning of the string.
• **width** (*float*) – Maximum width to use.

**Return type** int

**Returns** the index of text which will be used as the breakpoint, or `from_index` if there is no valid breakpoint.

GlyphString. **get_subwidth** *(from_index, to_index)*

Return the width of a slice of this string.

**Parameters**

• **from_index** (*int*) – The start index of the string to measure.
• **to_index** (*int*) – The end index (exclusive) of the string to measure.

**Return type** float
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.

Constructor:

    __init__ (font, text='', x=0, y=0, z=0, color=(1, 1, 1, 1), width=None, halign='left',
             valign='baseline')
    Create displayable text.

    Parameters
    • font (Font) – Font to render the text in.
    • text (str) – Initial string to render.
    • x (float) – X coordinate of the left edge of the text.
    • y (float) – Y coordinate of the baseline of the text. If the text is word-wrapped, this refers
      to the first line of text.
    • z (float) – Z coordinate of the text plane.
    • color (4-tuple of float) – Color to render the text in. Alpha values can be specified in the
      fourth component.
    • width (float) – Width to limit the rendering to. Text will be word-wrapped if necessary.
    • halign (str) – Alignment of the text. See Text.halign for details.
    • valign (str) – Controls positioning of the text based on the y coordinate. One of BASE-
      LINE, BOTTOM, CENTER or TOP. Defaults to BASELINE.

Methods:

    draw()
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASELINE</td>
<td>Align the baseline of the first line of text with the given Y coordinate.</td>
</tr>
<tr>
<td>CENTER</td>
<td>Align the bottom of the descender of the final line of text with the given Y coordinate.</td>
</tr>
<tr>
<td>LEFT</td>
<td>Align the horizontal center of the text to the given X coordinate.</td>
</tr>
<tr>
<td>RIGHT</td>
<td>Align the right edge of the text to the given X coordinate.</td>
</tr>
<tr>
<td>TOP</td>
<td>Align the top of the ascender of the first line of text with the given Y coordinate.</td>
</tr>
<tr>
<td>color</td>
<td>Text.<strong>color</strong></td>
</tr>
<tr>
<td>font</td>
<td>Text.<strong>font</strong></td>
</tr>
<tr>
<td>halign</td>
<td>Horizontal alignment of the text.</td>
</tr>
<tr>
<td>height</td>
<td>Height of the text.</td>
</tr>
<tr>
<td>leading</td>
<td>Vertical space between adjacent lines, in pixels.</td>
</tr>
<tr>
<td>line_height</td>
<td>Vertical distance between adjacent baselines, in pixels.</td>
</tr>
<tr>
<td>text</td>
<td>Text to render.</td>
</tr>
<tr>
<td>valign</td>
<td>Vertical alignment of the text.</td>
</tr>
<tr>
<td>width</td>
<td>Width of the text.</td>
</tr>
</tbody>
</table>

**Methods**

Text.**draw()**

**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**

**Functions**

- **add_directory(dir)**: Add a directory of fonts to pyglet’s search path.
- **add_file(font)**: Add a font to pyglet’s search path.
- **have_font(name)**: Check if specified system font name is available.
- **load([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 **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 load this file after adding it you must specify the face name to `load`, not the filename.

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

`compat_platform = ‘linux2’`
`str(object=’‘) -> string`
Return a nice string representation of the object. If the argument is a string, the return value is the same object.

`image = <pyglet._ModuleProxy object>`

`window = <pyglet._ModuleProxy object>`
Notes

Defined

• base
• gl
• glext_arb
• glu
• lib
• lib_glx
• math
• 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:

```python
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.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 Documentation, Release 1.2.4**

**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**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gl</code></td>
<td>Wrapper for /usr/include/GL/gl.h</td>
</tr>
<tr>
<td><code>gl_info</code></td>
<td>Information about version and extensions of current GL implementation.</td>
</tr>
<tr>
<td><code>glu</code></td>
<td>Wrapper for /usr/include/GL/glu.h</td>
</tr>
<tr>
<td><code>glu_info</code></td>
<td>Information about version and extensions of current GLU implementation.</td>
</tr>
</tbody>
</table>

**pyglet.gl.gl**  Wrapper for /usr/include/GL/gl.h

Generated by tools/gengl.py. Do not modify this file.

**Variables**

**DEFAULT_MODE = 0**

```python
int(x=0) -> int or long int(x, base=10) -> int or long
```

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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. >>>

```python
int('0b100', base=0) 4
```

**Notes**

- **util**

**pyglet.gl.gl_info**  Information about version and extensions of current GL implementation.

Usage:

```python
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:

```python
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

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.

    Methods:

    get_extensions()  Get a list of available OpenGL extensions.
    get_renderer()  Determine the renderer string of the OpenGL context.
    get_vendor()  Determine the vendor string of the OpenGL context.
    get_version()  Get the current OpenGL version.
    have_extension(extension)  Determine if an OpenGL extension is available.
    have_version(major[, minor, release])  Determine if a version of OpenGL is supported.
    remove_active_context()  Store information for the currently active context.

    Attributes:

    extensions
    have_context
    renderer
    vendor
    version

    Methods
GLInfo.get_extensions()  Get a list of available OpenGL extensions.

    Returns  a list of the available extensions.

    Return type  list of str
GLInfo.get_renderer()  
Determine the renderer string of the OpenGL context.  
   Return type  str

GLInfo.get_vendor()  
Determine the vendor string of the OpenGL context.  
   Return type  str

GLInfo.get_version()  
Get the current OpenGL version.  
   Returns  the OpenGL version  
   Return type  str

GLInfo.have_extension(extension)  
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

GLInfo.have_version(major, minor=0, release=0)  
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.

GLInfo.remove_active_context()  

GLInfo.set_active_context()  
Store information for the currently active context.  
   This method is called automatically for the default context.

Attributes
GLInfo.extensions = set([])  
GLInfo.have_context = False  
GLInfo.renderer = “”  
GLInfo.vendor = “”  
GLInfo.version = ‘0.0.0’

   have_context()  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 0x7f90fc178750>>`
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 0x7f90fc178750>>`
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 0x7f90fc178750>>`
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 0x7f90fc178750>>`
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 0x7f90fc178750>>`
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 0x7f90fc178750>>`
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 0x7f90fc178750>>`

**set_active_context** = `<bound method GLInfo.set_active_context of <pyglet.gl.gl_info.GLInfo object at 0x7f90fc178750>>`
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:
```python
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:

```python
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

**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.

Methods:
**get_extensions()**
Get a list of available GLU extensions.

**get_version()**
Get the current GLU version.

**have_extension(extension)**
Determine if a GLU extension is available.

**have_version(major[, minor, release])**
Determine if a version of GLU is supported.

**set_active_context()**
Store information for the currently active context.

---

**Attributes:**

- **extensions**
- **have_context**
- **version**

---

**Methods**

**GLUInfo.get_extensions()**
Get a list of available GLU extensions.

Returns a list of the available extensions.

Return type list of str

**GLUInfo.get_version()**
Get the current GLU version.

Returns the GLU version

Return type str

**GLUInfo.have_extension(extension)**
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

**GLUInfo.have_version(major, minor=0, release=0)**
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.

**GLUInfo.set_active_context()**
Store information for the currently active context.

This method is called automatically for the default context.
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 0x7f90fc186310>>
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 0x7f90fc186310>>
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 0x7f90fc186310>>
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 0x7f90fc186310>>
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 0x7f90fc186310>>
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

_ctypes.Structure → pyglet.gl.lib.c_void

c_void Class

class c_void

Attributes:

   dummy Type: CField

Attributes
c_void.dummy
Structure/Union member

GLException

Exception defined in pyglet.gl.lib

expection GLException

MissingFunctionException

Exception defined in pyglet.gl.lib

2.1. pyglet
exception MissingFunctionException (name, requires=None, suggestions=None)

decorate_function (func, name)
errcheck (result, func, arguments)
errcheck glBegin (result, func, arguments)
errcheck glEnd (result, func, arguments)
missing_function (name, requires=None, suggestions=None)

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 glBegin Function  Defined in pyglet.gl.lib
errcheck glBegin (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

Notes
  • pyglet

Classes

Exceptions

ConfigException

Continued on next page


### ContextException

Exception defined in `pyglet.gl` 

**exception** ContextException

---

### ConfigException

Exception defined in `pyglet.gl` 

**exception** ConfigException

---

**Functions**

- `get_current_context()` Return the active OpenGL context.

**get_current_context Function** Defined in `pyglet.gl`

```python
def 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**

- `compat_platform = ‘linux2’`

  `str(object='') -> string`

  Return a nice string representation of the object. If the argument is a string, the return value is the same object.


**Notes**

Defined

- glext_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:

```python
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:

```python
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:

```python
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:

```python
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.

```python
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

<table>
<thead>
<tr>
<th>allocation</th>
<th>Memory allocation algorithm for vertex arrays and buffers.</th>
</tr>
</thead>
</table>

Continued on next page
**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.

---

### Allocator Class

**class** `Allocator` *(capacity)*

Buffer space allocation implementation.

**Constructor:**

```python
__init__(capacity)
```

Create an allocator for a buffer of the specified capacity.

**Parameters** `capacity` *(int)* – Maximum size of the buffer.

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>alloc(size)</code></td>
<td>Allocate memory in the buffer.</td>
</tr>
<tr>
<td><code>dealloc(start, size)</code></td>
<td>Free a region of the buffer.</td>
</tr>
<tr>
<td><code>get_allocated_regions()</code></td>
<td>Get a list of (aggregate) allocated regions.</td>
</tr>
<tr>
<td><code>get_fragmentation()</code></td>
<td>Return fraction of free space that is not expandable.</td>
</tr>
<tr>
<td><code>get_fragmented_free_size()</code></td>
<td>Returns the amount of space unused, not including the final free block.</td>
</tr>
<tr>
<td><code>get_free_size()</code></td>
<td>Return the amount of space unused.</td>
</tr>
<tr>
<td><code>get_usage()</code></td>
<td>Return fraction of capacity currently allocated.</td>
</tr>
<tr>
<td><code>realloc(start, size, new_size)</code></td>
<td>Reallocate a region of the buffer.</td>
</tr>
<tr>
<td><code>set_capacity(size)</code></td>
<td>Resize the maximum buffer size.</td>
</tr>
</tbody>
</table>

---

**Methods**
Allocator.alloc(size)
Allocate memory in the buffer.
Raises AllocatorMemoryWarning if the allocation cannot be fulfilled.

Parameters size (int) – Size of region to allocate.
Return type int
Returns Starting index of the allocated region.

Allocator.dealloc(start, size)
Free a region of the buffer.

Parameters
• start (int) – Starting index of the region.
• size (int) – Size of the region.

Allocator.get_allocated_regions()
Get a list of (aggregate) allocated regions.

The result of this method is (starts, sizes), where starts is a list of starting indices of the regions and sizes their corresponding lengths.

Return type (list, list)

Allocator.get_fragmentation()
Return fraction of free space that is not expandable.

Return type float

Allocator.get_fragmented_free_size()
Returns the amount of space unused, not including the final free block.

Return type int

Allocator.get_free_size()
Return the amount of space unused.

Return type int

Allocator.get_usage()
Return fraction of capacity currently allocated.

Return type float

Allocator.realloc(start, size, new_size)
Reallocate a region of the buffer.
This is more efficient than separate dealloc and alloc calls, as the region can often be resized in-place.
Raises AllocatorMemoryWarning if the allocation cannot be fulfilled.

Parameters
• start (int) – Current starting index of the region.
• size (int) – Current size of the region.
• new_size (int) – New size of the region.

Allocator.set_capacity(size)
Resize the maximum buffer size.
The capacity cannot be reduced.

Parameters size (int) – New maximum size of the buffer.
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).

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:

\[
\text{attribute} ::= (\text{name} | \text{index} \ 'g' \ 'n'? | \text{texture} \ 't' \ ?) \ \text{count} \ \text{type}
\]

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.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractAttribute</td>
<td>Abstract accessor for an attribute in a mapped buffer.</td>
</tr>
<tr>
<td>ColorAttribute</td>
<td>Color vertex attribute.</td>
</tr>
<tr>
<td>EdgeFlagAttribute</td>
<td>Edge flag attribute.</td>
</tr>
<tr>
<td>FogCoordAttribute</td>
<td>Fog coordinate attribute.</td>
</tr>
<tr>
<td>GenericAttribute</td>
<td>Generic vertex attribute, used by shader programs.</td>
</tr>
<tr>
<td>MultiTexCoordAttribute</td>
<td>Texture coordinate attribute.</td>
</tr>
<tr>
<td>NormalAttribute</td>
<td>Normal vector attribute.</td>
</tr>
<tr>
<td>SecondaryColorAttribute</td>
<td>Secondary color attribute.</td>
</tr>
<tr>
<td>TexCoordAttribute</td>
<td>Texture coordinate attribute.</td>
</tr>
<tr>
<td>VertexAttribute</td>
<td>Vertex coordinate attribute.</td>
</tr>
</tbody>
</table>

Classes
AbstractAttribute Class

class AbstractAttribute (count, gl_type)
    Abstract accessor for an attribute in a mapped buffer.

    Constructor:
    __init__ (count, gl_type)
        Create the attribute accessor.

        Parameters
        • count (int) – Number of components in the attribute.
        • gl_type (int) – OpenGL type enumerant; for example, GL_FLOAT

        Methods:
        enable ()
            Enable the attribute using glEnableClientState.
        get_region (buffer, start, count)
            Map a buffer region using this attribute as an accessor.
            The returned region consists of a contiguous array of component data elements. For example, if this
ttribute uses 3 floats per vertex, and the count parameter is 4, the number of floats mapped will be 3 * 4 = 12.
            Parameters
            • buffer (AbstractMappable) – The buffer to map.
            • start (int) – Offset of the first vertex to map.
            • count (int) – Number of vertices to map
            Return type AbstractBufferRegion
        set_pointer (offset)
            Setup this attribute to point to the currently bound buffer at the given offset.
            offset should be based on the currently bound buffer’s ptr member.
            Parameters offset (int) – Pointer offset to the currently bound buffer for this attribute.
AbstractAttribute.\texttt{set\_region}(buffer, start, count, data)
Set the data over a region of the buffer.

Parameters

- \textbf{buffer} (AbstractMappable) – The buffer to modify.
- \textbf{start} (int) – Offset of the first vertex to set.
- \textbf{count} (int) – Number of vertices to set.
- \textbf{data} (sequence) – Sequence of data components.

\begin{center}
pyglet.graphics.vertexattribute.AbstractAttribute $\rightarrow$ pyglet.graphics.vertexattribute.ColorAttribute
\end{center}

\textbf{ColorAttribute} Class

\texttt{class ColorAttribute(count, gl\_type)}
Color vertex attribute.

\textbf{Constructor:}

\texttt{\_\_init\_\_}(count, gl\_type)

\textbf{Methods:}

\begin{itemize}
\item \texttt{enable()}
\item \texttt{get\_region(buffer, start, count)} Map a buffer region using this attribute as an accessor.
\item \texttt{set\_pointer(pointer)}
\item \texttt{set\_region(buffer, start, count, data)} Set the data over a region of the buffer.
\end{itemize}

\textbf{Attributes:}

\begin{itemize}
\item \texttt{plural}
\end{itemize}

\textbf{Methods}

\texttt{ColorAttribute\_enable()}\texttt{ColorAttribute\_set\_pointer(pointer)}

\textbf{Attributes}

\texttt{ColorAttribute.plural = 'colors'}

\textbf{Inherited members}
Methods

ColorAttribute.get_region(buffer, start, count)
Map a buffer region using this attribute as an accessor.

The returned region can be modified as if the buffer was a contiguous array of this attribute (though it may actually be interleaved or otherwise non-contiguous).

The returned region consists of a contiguous array of component data elements. For example, if this attribute uses 3 floats per vertex, and the count parameter is 4, the number of floats mapped will be $3 \times 4 = 12$.

Parameters

- **buffer** (AbstractMappable) – The buffer to map.
- **start** (int) – Offset of the first vertex to map.
- **count** (int) – Number of vertices to map

Return type: AbstractBufferRegion

ColorAttribute.set_region(buffer, start, count, data)
Set the data over a region of the buffer.

Parameters

- **buffer** (AbstractMappable) – The buffer to modify.
- **start** (int) – Offset of the first vertex to set.
- **count** (int) – Number of vertices to set.
- **data** (sequence) – Sequence of data components.

---

EdgeFlagAttribute Class

class EdgeFlagAttribute (gl_type)
Edge flag attribute.

Constructor:

__init__(gl_type)

Methods:

- **enable()**
- get_region(buffer, start, count) Map a buffer region using this attribute as an accessor.
- set_pointer(pointer)
- set_region(buffer, start, count, data) Set the data over a region of the buffer.

Attributes:
Methods
EdgeFlagAttribute.enable()
EdgeFlagAttribute.set_pointer(pointer)

Attributes
EdgeFlagAttribute.plural = 'edge_flags'

Inherited members

Methods

EdgeFlagAttribute.get_region(buffer, start, count)
Map a buffer region using this attribute as an accessor.

The returned region can be modified as if the buffer was a contiguous array of this attribute (though it may actually be interleaved or otherwise non-contiguous).

The returned region consists of a contiguous array of component data elements. For example, if this attribute uses 3 floats per vertex, and the count parameter is 4, the number of floats mapped will be $3 \times 4 = 12$.

Parameters

- buffer (AbstractMappable) – The buffer to map.
- start (int) – Offset of the first vertex to map.
- count (int) – Number of vertices to map

Return type AbstractBufferRegion

EdgeFlagAttribute.set_region(buffer, start, count, data)
Set the data over a region of the buffer.

Parameters

- buffer (AbstractMappable) – The buffer to modify.
- start (int) – Offset of the first vertex to set.
- count (int) – Number of vertices to set.
- data (sequence) – Sequence of data components.

FogCoordAttribute Class
class FogCoordAttribute(count, gl_type)
    Fog coordinate attribute.

    Constructor:
    __init__(count, gl_type)

    Methods:

    enable()  
    get_region(buffer, start, count) Map a buffer region using this attribute as an accessor.
    set_pointer(pointer)
    set_region(buffer, start, count, data) Set the data over a region of the buffer.

    Attributes:

    plural

Inherited members

Methods
FogCoordAttribute.enable()  
FogCoordAttribute.set_pointer(pointer)

Attributes
FogCoordAttribute.plural = ‘fog_coords’

The returned region can be modified as if the buffer was a contiguous array of this attribute (though it may actually be interleaved or otherwise non-contiguous).

The returned region consists of a contiguous array of component data elements. For example, if this attribute uses 3 floats per vertex, and the count parameter is 4, the number of floats mapped will be 3 * 4 = 12.

Parameters

- buffer (AbstractMappable) – The buffer to map.
- start (int) – Offset of the first vertex to map.
- count (int) – Number of vertices to map

Return type AbstractBufferRegion

FogCoordAttribute.set_region(buffer, start, count, data)
Set the data over a region of the buffer.
Parameters

- **buffer** *(AbstractMappable)* – The buffer to modify.
- **start** *(int)* – Offset of the first vertex to set.
- **count** *(int)* – Number of vertices to set.
- **data** *(sequence)* – Sequence of data components.

**GenericAttribute** Class

class **GenericAttribute** *(index, normalized, count, gl_type)*

Generic vertex attribute, used by shader programs.

**Constructor:**

```
__init__(index, normalized, count, gl_type)
```

**Methods:**

- **enable()**
- **get_region(buffer, start, count)** – Map a buffer region using this attribute as an accessor.
- **set_pointer(pointer)**
- **set_region(buffer, start, count, data)** – Set the data over a region of the buffer.

**Methods**

- **GenericAttribute.enable()**
- **GenericAttribute.set_pointer(pointer)**

**Inherited members**

**Methods**

- **genericAttrtribute.get_region(buffer, start, count)**
  
  Map a buffer region using this attribute as an accessor.

  The returned region can be modified as if the buffer was a contiguous array of this attribute (though it may actually be interleaved or otherwise non-contiguous).

  The returned region consists of a contiguous array of component data elements. For example, if this attribute uses 3 floats per vertex, and the `count` parameter is 4, the number of floats mapped will be 3 * 4 = 12.

**Parameters**

- **buffer** *(AbstractMappable)* – The buffer to map.
- **start** *(int)* – Offset of the first vertex to map.
GenericAttribute.set_region(buffer, start, count, data)

Set the data over a region of the buffer.

Parameters

• buffer (AbstractMappable) – The buffer to modify.
• start (int) – Offset of the first vertex to set.
• count (int) – Number of vertices to set.
• data (sequence) – Sequence of data components.

MultiTexCoordAttribute Class
class MultiTexCoordAttribute (texture, count, gl_type)

Texture coordinate attribute.

Constructor:

__init__ (texture, count, gl_type)

Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable()</td>
<td></td>
</tr>
<tr>
<td>get_region(buffer, start, count)</td>
<td>Map a buffer region using this attribute as an accessor.</td>
</tr>
<tr>
<td>set_pointer(pointer)</td>
<td></td>
</tr>
<tr>
<td>set_region(buffer, start, count, data)</td>
<td>Set the data over a region of the buffer.</td>
</tr>
</tbody>
</table>

Methods

MultiTexCoordAttribute.enable()

MultiTexCoordAttribute.set_pointer(pointer)

Inherited members

Methods

MultiTexCoordAttribute.get_region(buffer, start, count)

Map a buffer region using this attribute as an accessor.

The returned region can be modified as if the buffer was a contiguous array of this attribute (though it may actually be interleaved or otherwise non-contiguous).

The returned region consists of a contiguous array of component data elements. For example, if this attribute uses 3 floats per vertex, and the count parameter is 4, the number of floats mapped will be $3 \times 4 = 12$. 
Parameters

- **buffer** (AbstractMappable) – The buffer to map.
- **start** (int) – Offset of the first vertex to map.
- **count** (int) – Number of vertices to map

Return type: AbstractBufferRegion

MultiTexCoordAttribute.

set_region (buffer, start, count, data)

Set the data over a region of the buffer.

Parameters

- **buffer** (AbstractMappable) – The buffer to modify.
- **start** (int) – Offset of the first vertex to set.
- **count** (int) – Number of vertices to set.
- **data** (sequence) – Sequence of data components.

---

**NormalAttribute Class**

class NormalAttribute (gl_type)

Normal vector attribute.

Constructor:

`__init__(gl_type)`

Methods:

- `enable()`
- `get_region(buffer, start, count)` Map a buffer region using this attribute as an accessor.
- `set_pointer(pointer)`
- `set_region(buffer, start, count, data)` Set the data over a region of the buffer.

Attributes:

- `plural`
NormalAttribute.plural = ‘normals’

Inherited members

Methods

```
NormalAttribute.get_region(buffer, start, count)
Map a buffer region using this attribute as an accessor.

The returned region can be modified as if the buffer was a contiguous array of this attribute (though
it may actually be interleaved or otherwise non-contiguous).

The returned region consists of a contiguous array of component data elements. For example, if this
attribute uses 3 floats per vertex, and the count parameter is 4, the number of floats mapped will be
3 * 4 = 12.

Parameters

* buffer (AbstractMappable) – The buffer to map.
* start (int) – Offset of the first vertex to map.
* count (int) – Number of vertices to map

Return type AbstractBufferRegion

NormalAttribute.set_region(buffer, start, count, data)
Set the data over a region of the buffer.

Parameters

* buffer (AbstractMappable) – The buffer to modify.
* start (int) – Offset of the first vertex to set.
* count (int) – Number of vertices to set.
* data (sequence) – Sequence of data components.
```

SecondaryColorAttribute Class

```
class SecondaryColorAttribute (gl_type)
Secondary color attribute.

Constructor:

__init__ (gl_type)

Methods:

enable()
get_region(buffer, start, count) Map a buffer region using this attribute as an accessor.
set_pointer(pointer)
set_region(buffer, start, count, data) Set the data over a region of the buffer.
```
Attributes:

```
plural
```

Methods

SecondaryColorAttribute.enable()
SecondaryColorAttribute.set_pointer(pointer)

Attributes

SecondaryColorAttribute.plural = 'secondary_colors'

Inherited members

Methods

SecondaryColorAttribute.get_region(buffer, start, count)

Map a buffer region using this attribute as an accessor.

The returned region can be modified as if the buffer was a contiguous array of this attribute (though it may actually be interleaved or otherwise non-contiguous).

The returned region consists of a contiguous array of component data elements. For example, if this attribute uses 3 floats per vertex, and the `count` parameter is 4, the number of floats mapped will be $3 \times 4 = 12$.

Parameters

- `buffer (AbstractMappable)` – The buffer to map.
- `start (int)` – Offset of the first vertex to map.
- `count (int)` – Number of vertices to map

Return type `AbstractBufferRegion`

SecondaryColorAttribute.set_region(buffer, start, count, data)

Set the data over a region of the buffer.

Parameters

- `buffer (AbstractMappable)` – The buffer to modify.
- `start (int)` – Offset of the first vertex to set.
- `count (int)` – Number of vertices to set.
- `data (sequence)` – Sequence of data components.
**TexCoordAttribute Class**

```python
class TexCoordAttribute (count, gl_type)
```

Texture coordinate attribute.

**Constructor:**

```python
__init__(count, gl_type)
```

**Methods:**

- `convert_to_multi_tex_coord_attribute()` Changes the class of the attribute to `MultiTexCoordAttribute`.
- `enable()`
- `get_region(buffer, start, count)` Map a buffer region using this attribute as an accessor.
- `set_pointer(pointer)`
- `set_region(buffer, start, count, data)` Set the data over a region of the buffer.

**Attributes:**

```python
plural
```

**Methods**

- `TexCoordAttribute.convert_to_multi_tex_coord_attribute()` Changes the class of the attribute to `MultiTexCoordAttribute`.
- `TexCoordAttribute.enable()`
- `TexCoordAttribute.set_pointer(pointer)`

**Attributes**

- `TexCoordAttribute.plural = 'tex_coords'`

**Inherited members**

**Methods**

- `TexCoordAttribute.get_region(buffer, start, count)` Map a buffer region using this attribute as an accessor.

  The returned region can be modified as if the buffer was a contiguous array of this attribute (though it may actually be interleaved or otherwise non-contiguous).

  The returned region consists of a contiguous array of component data elements. For example, if this attribute uses 3 floats per vertex, and the `count` parameter is 4, the number of floats mapped will be $3 \times 4 = 12$. 

164 Chapter 2. API Reference
Parameters

- **buffer** (*AbstractMappable*) – The buffer to map.
- **start** (*int*) – Offset of the first vertex to map.
- **count** (*int*) – Number of vertices to map

**Return type** *AbstractBufferRegion*

**TexCoordAttribute**.**set_region**(buffer, start, count, data)
Set the data over a region of the buffer.

- **buffer** (*AbstractMappable*) – The buffer to modify.
- **start** (*int*) – Offset of the first vertex to set.
- **count** (*int*) – Number of vertices to set.
- **data** (*sequence*) – Sequence of data components.

```
pyglet.graphics.vertexattribute.AbstractAttribute ➔ pyglet.graphics.vertexattribute.VertexAttribute
```

**VertexAttribute** Class

class **VertexAttribute**(count, gl_type)
Vertex coordinate attribute.

**Constructor:**

`__init__`(count, gl_type)

**Methods:**

- `enable()`
- `get_region(buffer, start, count)` Map a buffer region using this attribute as an accessor.
- `set_pointer(pointer)`
- `set_region(buffer, start, count, data)` Set the data over a region of the buffer.

**Attributes:**

- `plural`

```
Methods
VertexAttribute.enable()
VertexAttribute.set_pointer(pointer)
```

**Attributes**

2.1. pyglet
VertexAttribute\text{.plural} = \textquote{vertices}'

Inherited members

Methods

\texttt{getAttribute} (\textit{buffer, start, count})
Map a buffer region using this attribute as an accessor.

The returned region can be modified as if the buffer was a contiguous array of this attribute (though it may actually be interleaved or otherwise non-contiguous).

The returned region consists of a contiguous array of component data elements. For example, if this attribute uses 3 floats per vertex, and the \textit{count} parameter is 4, the number of floats mapped will be $3 \times 4 = 12$.

Parameters

- \texttt{buffer} (AbstractMappable) – The buffer to map.
- \texttt{start} (int) – Offset of the first vertex to map.
- \texttt{count} (int) – Number of vertices to map

Return type \texttt{AbstractBufferRegion}

\texttt{setRegion} (\textit{buffer, start, count, data})
Set the data over a region of the buffer.

Parameters

- \texttt{buffer} (AbstractMappable) – The buffer to modify.
- \texttt{start} (int) – Offset of the first vertex to set.
- \texttt{count} (int) – Number of vertices to set.
- \texttt{data} (sequence) – Sequence of data components.

\begin{tabular}{l}
\texttt{create_attribute} (format) Create a vertex attribute description from a format string. \\
\texttt{interleave_attributes} (attributes) Interleave attribute offsets. \\
\texttt{serialize_attributes} (count, attributes) Serialize attribute offsets.
\end{tabular}

Functions

\texttt{create_attribute} Function Defined in \texttt{pyglet.graphics.vertexattribute}

\texttt{create_attribute} (format)
Create a vertex attribute description from a format string.

The initial stride and offset of the attribute will be 0.

Parameters \texttt{format} (str) – Attribute format string. See the module summary for details.

Return type \texttt{AbstractAttribute}

\texttt{interleave_attributes} Function Defined in \texttt{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
compat_platform = ‘linux2’
str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

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.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractBuffer</td>
<td>Abstract buffer of byte data.</td>
</tr>
<tr>
<td>AbstractBufferRegion</td>
<td>A mapped region of a buffer.</td>
</tr>
<tr>
<td>AbstractMappable</td>
<td></td>
</tr>
<tr>
<td>IndirectArrayRegion</td>
<td>A mapped region in which data elements are not necessarily contiguous.</td>
</tr>
<tr>
<td>MappableVertexBufferObject</td>
<td>A VBO with system-memory backed store.</td>
</tr>
<tr>
<td>VertexArray</td>
<td>A ctypes implementation of a vertex array.</td>
</tr>
<tr>
<td>VertexArrayRegion</td>
<td>A mapped region of a vertex array.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 2.78 – continued from previous page

<table>
<thead>
<tr>
<th>Class name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VertexBufferObject</td>
<td>Lightweight representation of an OpenGL VBO.</td>
</tr>
<tr>
<td>VertexBufferObjectRegion</td>
<td>A mapped region of a VBO.</td>
</tr>
</tbody>
</table>

Classes

```python
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`

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bind()</code></td>
<td>Bind this buffer to its OpenGL target.</td>
</tr>
<tr>
<td><code>delete()</code></td>
<td>Delete this buffer, reducing system resource usage.</td>
</tr>
<tr>
<td><code>map([invalidate])</code></td>
<td>Map the entire buffer into system memory.</td>
</tr>
<tr>
<td><code>resize(size)</code></td>
<td>Resize the buffer to a new size.</td>
</tr>
<tr>
<td><code>set_data(data)</code></td>
<td>Set the entire contents of the buffer.</td>
</tr>
<tr>
<td><code>set_data_region(data, start, length)</code></td>
<td>Set part of the buffer contents.</td>
</tr>
<tr>
<td><code>unbind()</code></td>
<td>Reset the buffer’s OpenGL target.</td>
</tr>
<tr>
<td><code>unmap()</code></td>
<td>Unmap a previously mapped memory block.</td>
</tr>
</tbody>
</table>

**Attributes:**

```
ptr
size
```

**Methods**

- AbstractBuffer.`bind()`
  
  Bind this buffer to its OpenGL target.

- AbstractBuffer.`delete()`
  
  Delete this buffer, reducing system resource usage.
AbstractBuffer.map(invalidate=False)
Map the entire buffer into system memory.

The mapped region must be subsequently unmapped with unmap before performing any other operations on the buffer.

**Parameters**

- invalidate (bool) – If True, the initial contents of the mapped block need not reflect the actual contents of the buffer.

**Return type**

POINTER(ctypes.c_ubyte)

**Returns**

Pointer to the mapped block in memory

AbstractBuffer.resize(size)
Resize the buffer to a new size.

**Parameters**

- size (int) – New size of the buffer, in bytes

AbstractBuffer.set_data(data)
Set the entire contents of the buffer.

**Parameters**

- data (sequence of int or ctypes pointer) – The byte array to set

AbstractBuffer.set_data_region(data, start, length)
Set part of the buffer contents.

**Parameters**

- data (sequence of int or ctypes pointer) – The byte array of data to set
- start (int) – Offset to start replacing data
- length (int) – Length of region to replace

AbstractBuffer.unbind()
Reset the buffer’s OpenGL target.

AbstractBuffer.unmap()
Unmap a previously mapped memory block.

**Attributes**

AbstractBuffer.ptr = 0
AbstractBuffer.size = 0

---

**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.

**Methods**

---

2.1. pyglet 169
Methods

AbstractBufferRegion.\texttt{invalidate}()

Mark this region as changed.

The buffer may not be updated with the latest contents of the array until this method is called. (However, it may not be updated until the next time the buffer is used, for efficiency).

\begin{Verbatim}
pyglet.graphics.vertexbuffer.AbstractMappable
\end{Verbatim}

\textit{AbstractMappable} Class

class AbstractMappable
Methods:

\begin{Verbatim}
get_region\phantom{deployment}(start, size, ptr_type) Map a region of the buffer into a ctypes array of the desired type.
\end{Verbatim}

Methods

AbstractMappable.\texttt{get_region}(\textit{start, size, ptr_type})

Map a region of the buffer into a ctypes array of the desired type. This region does not need to be unmapped, but will become invalid if the buffer is resized.

Note that although a pointer type is required, an array is mapped. For example:

\begin{Verbatim}
get_region(0, ctypes.sizeof(c_int) * 20, ctypes.POINTER(c_int * 20))
\end{Verbatim}

will map bytes 0 to 80 of the buffer to an array of 20 ints.

Changes to the array may not be recognised until the region’s AbstractBufferRegion.invalidate method is called.

\begin{itemize}
\item \texttt{start} (int) – Offset into the buffer to map from, in bytes
\item \texttt{size} (int) – Size of the buffer region to map, in bytes
\item \texttt{ptr_type} (ctypes pointer type) – Pointer type describing the array format to create
\end{itemize}

\textbf{Return type} AbstractBufferRegion
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.

Constructor:

__init__ (region, size, component_count, component_stride)

Wrap a buffer region.

Use the component_count and component_stride parameters to specify the data layout of the encapsulated region. For example, if RGBA data is to be accessed as if it were packed RGB, component_count would be set to 3 and component_stride to 4. If the region contains 10 RGBA tuples, the size parameter is $3 \times 10 = 30$.

Parameters

• region (AbstractBufferRegion) – The region with interleaved data

• size (int) – The number of elements that this region will provide access to.

• component_count (int) – The number of elements that are contiguous before some must be skipped.

• component_stride (int) – The number of elements of interleaved data separating the contiguous sections.

Methods:

 invalidate()
There may also be less performance penalty for resizing this buffer.
Updates to data via `map` are committed immediately.

**Constructor:**

```
__init__(size, target, usage)
```

**Methods:**

```
bind()
delete()
get_region(start, size, ptr_type)
map([invalidate])
resize(size)
set_data(data)
set_data_region(data, start, length)
unbind()
unmap()
```

**Attributes:**

```
ptr
size
```

**Methods**

1. `MappableVertexBufferObject.bind()`
2. `MappableVertexBufferObject.get_region(start, size, ptr_type)`
3. `MappableVertexBufferObject.map(invalidate=False)`
4. `MappableVertexBufferObject.resize(size)`
5. `MappableVertexBufferObject.set_data(data)`
6. `MappableVertexBufferObject.set_data_region(data, start, length)`
7. `MappableVertexBufferObject.unbind()`
8. `MappableVertexBufferObject.unmap()`

**Inherited members**

**Methods**

1. `MappableVertexBufferObject.delete()`
2. `MappableVertexBufferObject.unbind()`

**Attributes**

1. `MappableVertexBufferObject.ptr = 0`
2. `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, unmmap 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.

**Constructor:**

__init__(size)

**Methods:**

bind()
delete()
get_region(start, size, ptr_type)
map([invalidate])
resize(size)
set_data(data)
set_data_region(data, start, length)
unbind()
unmap()

**Attributes:**

ptr
size

Methods
VertexArray.bind()
VertexArray.delete()
VertexArray.get_region (start, size, ptr_type)
VertexArray.map (invalidate=False)
VertexArray.resize(size)
VertexArray.set_data(data)
VertexArray.set_data_region (data, start, length)
VertexArray.unbind()
VertexArray.unmap()

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.

Constructor:
__init__ (array)

Methods:

invalidated() Mark this region as changed.

Inherited members

Methods

VertexArrayRegion.invalidate() Mark this region as changed.

The buffer may not be updated with the latest contents of the array until this method is called. (However, it may not be updated until the next time the buffer is used, for efficiency).
**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`.

**Constructor:**

`__init__`(size, target, usage)

**Methods:**

- `bind()`
- `delete()`
- `map([invalidate])`
- `resize(size)`
- `set_data(data)`
- `set_data_region(data, start, length)`
- `unbind()`
- `unmap()`

**Attributes:**

- `ptr`
- `size`
**VertexBufferObjectRegion**  

class **VertexBufferObjectRegion** *(buffer, start, end, array)*  

A mapped region of a VBO.  

**Constructor:**  

```python
__init__(buffer, start, end, array)
```

**Methods:**  

```python
invalidate()
```

**Methods**  

**VertexBufferObjectRegion**. `invalidate()`

---

**create_buffer** Function  

Defined in `pyglet.graphics.vertexbuffer`

```python
create_buffer(size[, target, usage, vbo])  
```

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`

```python
create_mappable_buffer(size[, target, ...])  
```

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**

```python
compat_platform = ‘linux2’
str(object='') -> string
```

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

**Defined**

- *gl*
- *glext_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.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndexedVertexDomain</td>
<td>Management of a set of indexed vertex lists.</td>
</tr>
<tr>
<td>IndexedVertexList</td>
<td>A list of vertices within an <em>IndexedVertexDomain</em> that are indexed.</td>
</tr>
<tr>
<td>VertexDomain</td>
<td>A list of vertices within an <em>IndexedVertexDomain</em>.</td>
</tr>
<tr>
<td>VertexList</td>
<td>Management of a set of vertex lists.</td>
</tr>
</tbody>
</table>

**Classes**

pyglet.graphics.vertexdomain.VertexDomain → pyglet.graphics.vertexdomain.IndexedVertexDomain
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.

**Constructor:**

```
__init__(attribute_usages, index_gl_type=5125)
```

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create(count, index_count)</code></td>
<td>Create an <code>IndexedVertexList</code> in this domain.</td>
</tr>
<tr>
<td><code>draw(mode[, vertex_list])</code></td>
<td>Draw vertices in the domain.</td>
</tr>
<tr>
<td><code>get_index_region(start, count)</code></td>
<td>Get a region of the index buffer.</td>
</tr>
</tbody>
</table>

### Methods

**IndexedVertexDomain**. `create` *(count, index_count)*

Create an `IndexedVertexList` in this domain.

**Parameters**

- `count (int)` – Number of vertices to create
- `index_count` – Number of indices to create

**IndexedVertexDomain**. `draw` *(mode, vertex_list=None)*

Draw vertices in the domain.

If `vertex_list` is not specified, all vertices in the domain are drawn. This is the most efficient way to render primitives.

If `vertex_list` specifies a `VertexList`, only primitives in that list will be drawn.

**Parameters**

- `mode (int)` – OpenGL drawing mode, e.g. GL_POINTS, GL_LINES, etc.
- `vertex_list` *(IndexedVertexList)* – Vertex list to draw, or `None` for all lists in this domain.

**IndexedVertexDomain**. `get_index_region` *(start, count)*

Get a region of the index buffer.

**Parameters**

- `start (int)` – Start of the region to map.
- `count (int)` – Number of indices to map.

**Return type** Array of int

---

**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.

Constructor:
__init__ (domain, start, count, index_start, index_count)

Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete()</td>
<td>Delete this group.</td>
</tr>
<tr>
<td>draw(mode)</td>
<td></td>
</tr>
<tr>
<td>get_domain()</td>
<td>Get the domain this vertex list belongs to.</td>
</tr>
<tr>
<td>get_size()</td>
<td>Get the number of vertices in the list.</td>
</tr>
<tr>
<td>migrate(domain)</td>
<td>Move this group from its current domain and add to the specified one.</td>
</tr>
<tr>
<td>resize(count, index_count)</td>
<td>Resize this group.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>colors</td>
<td>Array of color data.</td>
</tr>
<tr>
<td>edge_flags</td>
<td>Array of edge flag data.</td>
</tr>
<tr>
<td>fog_coords</td>
<td>Array of fog coordinate data.</td>
</tr>
<tr>
<td>indices</td>
<td>Array of index data.</td>
</tr>
<tr>
<td>multi_tex_coors</td>
<td>Multi-array texture coordinate data.</td>
</tr>
<tr>
<td>normals</td>
<td>Array of normal vector data.</td>
</tr>
<tr>
<td>secondary_colors</td>
<td>Array of secondary color data.</td>
</tr>
<tr>
<td>tex_coords</td>
<td>Array of texture coordinate data.</td>
</tr>
<tr>
<td>vertices</td>
<td>Array of vertex coordinate data.</td>
</tr>
</tbody>
</table>

Methods
IndexedVertexList.delete()
Delete this group.

IndexedVertexList.draw(mode)

IndexedVertexList.resize(count, index_count)
Resize this group.

Parameters

- count (int) – New number of vertices in the list.
- index_count (int) – New number of indices in the list.

Attributes
IndexedVertexList.indices
Array of index data.

Inherited members
Methods

IndexedVertexList.get_domain()
Get the domain this vertex list belongs to.

Return type VertexDomain

IndexedVertexList.get_size()
Get the number of vertices in the list.

Return type int

IndexedVertexList.migrate(domain)
Move this group from its current domain and add to the specified one. Attributes on domains must match. (In practice, used to change parent state of some vertices).

Parameters domain (VertexDomain) – Domain to migrate this vertex list to.

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.
__init__ (attribute_usages)

Methods:

create(count) Create a VertexList in this domain.
draw(mode[, vertex_list]) Draw vertices in the domain.

Methods

VertexDomain.create (count)
Create a VertexList in this domain.

Parameters count (int) – Number of vertices to create.

Return type VertexList

VertexDomain.draw (mode, vertex_list=None)
Draw vertices in the domain.

If vertex_list is not specified, all vertices in the domain are drawn. This is the most efficient way to render primitives.

If vertex_list specifies a VertexList, only primitives in that list will be drawn.

Parameters

• mode (int) – OpenGL drawing mode, e.g. GL_POINTS, GL_LINES, etc.
• vertex_list (VertexList) – Vertex list to draw, or None for all lists in this domain.

VertexList Class

class VertexList (domain, start, count)
A list of vertices within a VertexDomain. Use VertexDomain.create to construct this list.

Constructor:

__init__ (domain, start, count)

Methods:

delete() Delete this group.
draw(mode) Draw this vertex list in the given OpenGL mode.
get_domain() Get the domain this vertex list belongs to.
get_size() Get the number of vertices in the list.
migrate(domain) Move this group from its current domain and add to the specified one.
resize(count) Resize this group.

Attributes:
colors
Array of color data.

edge_flags
Array of edge flag data.

fog_coords
Array of fog coordinate data.

multi_tex_coords
Multi-array texture coordinate data.

normals
Array of normal vector data.

secondary_colors
Array of secondary color data.

tex_coords
Array of texture coordinate data.

vertices
Array of vertex coordinate data.

Methods

**VertexList.delete()**
Delete this group.

**VertexList.draw(mode)**
Draw this vertex list in the given OpenGL mode.

**Parameters**

*mode* (*int*) – OpenGL drawing mode, e.g. GL_POINTS, GL_LINES, etc.

**VertexList.get_domain()**
Get the domain this vertex list belongs to.

**Return type** *VertexDomain*

**VertexList.get_size()**
Get the number of vertices in the list.

**Return type** *int*

**VertexList.migrate(domain)**
Move this group from its current domain and add to the specified one. Attributes on domains must match. (In practice, used to change parent state of some vertices).

**Parameters**

*domain* (*VertexDomain*) – Domain to migrate this vertex list to.

**VertexList.resize(count)**
Resize this group.

**Parameters**

*count* (*int*) – New number of vertices in the list.

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.

create_attribute_usage(format)
   Create an attribute and usage pair from a format string.

create_domain(*attribute_usage_formats)
   Create a vertex domain covering the given attribute usage formats.

create_indexed_domain(*attribute_usage_formats)
   Create an indexed vertex domain covering the given attribute usage formats.

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 \texttt{static} also indicates a preference for interleaved arrays. If \texttt{none} is specified a buffer object is not created, and vertex data is stored in system memory.

   Some examples:

   \texttt{v3f/stream} 3D vertex position using floats, for stream usage
   \texttt{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 \texttt{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 \texttt{VertexDomain}

Variables

compat_platform = \texttt{\textquotesingle}linux2\textquotesingle
   str(object=\textquotesingle\textquotesingle) -> string

   Return a nice string representation of the object. If the argument is a string, the return value is the same object.
### Defined

- allocation
- gl
- glext_arb
- glu
- lib
- lib_glx
- re
- vertexattribute
- vertexbuffer

### Notes

#### Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Batch</strong></td>
<td>Manage a collection of vertex lists for batched rendering.</td>
</tr>
<tr>
<td><strong>Group</strong></td>
<td>Group of common OpenGL state.</td>
</tr>
<tr>
<td><strong>NullGroup</strong></td>
<td>The default group class used when None is given to a batch.</td>
</tr>
<tr>
<td><strong>OrderedGroup</strong></td>
<td>A group with partial order.</td>
</tr>
<tr>
<td><strong>TextureGroup</strong></td>
<td>A group that enables and binds a texture.</td>
</tr>
</tbody>
</table>

```
pyglet.graphics.Batch
```

**Batch Class**

```python
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.

**Constructor:**

```python
__init__()
Create a graphics batch.
```

**Methods:**

- `add(count, mode, group, *data)` Add a vertex list to the batch.
- `add_indexed(count, mode, group, indices, *data)` Add an indexed vertex list to the batch.
- `draw()` Draw the batch.
- `draw_subset(vertex_lists)` Draw only some vertex lists in the batch.
- `invalidate()` Force the batch to update the draw list.
- `migrate(vertex_list, mode, group, batch)` Migrate a vertex list to another batch and/or group.
Methods

Batch.add(count, mode, group, *data)

Add a vertex list to the batch.

Parameters

- **count** (int) – The number of vertices in the list.
- **mode** (int) – OpenGL drawing mode enumeration; for example, one of GL_POINTS, GL_LINES, GL_TRIANGLES, etc. See the module summary for additional information.
- **group** (Group) – Group of the vertex list, or None if no group is required.
- **data** (data items) – Attribute formats and initial data for the vertex list. See the module summary for details.

Return type VertexList

Batch.add_indexed(count, mode, group, indices, *data)

Add an indexed vertex list to the batch.

Parameters

- **count** (int) – The number of vertices in the list.
- **mode** (int) – OpenGL drawing mode enumeration; for example, one of GL_POINTS, GL_LINES, GL_TRIANGLES, etc. See the module summary for additional information.
- **group** (Group) – Group of the vertex list, or None if no group is required.
- **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

Batch.draw()

Draw the batch.

Batch.draw_subset(vertex_lists)

Draw only some vertex lists in the batch.

The use of this method is highly discouraged, as it is quite inefficient. Usually an application can be redesigned so that batches can always be drawn in their entirety, using draw.

The given vertex lists must belong to this batch; behaviour is undefined if this condition is not met.

Parameters vertex_lists (sequence of VertexList or IndexedVertexList) – Vertex lists to draw.

Batch.invalidate()

Force the batch to update the draw list.

This method can be used to force the batch to re-compute the draw list when the ordering of groups has changed.

Note: Since pyglet 1.2

Batch.migrate(vertex_list, mode, group, batch)

Migrate a vertex list to another batch and/or group.

vertex_list and mode together identify the vertex list to migrate. group and batch are new owners of the vertex list after migration.
The results are undefined if `mode` is not correct or if `vertex_list` does not belong to this batch (they are not checked and will not necessarily throw an exception immediately).

`batch` can remain unchanged if only a group change is desired.

**Parameters**

- **vertex_list** (`VertexList`) – A vertex list currently belonging to this batch.
- **mode** (`int`) – The current GL drawing mode of the vertex list.
- **group** (`Group`) – The new group to migrate to.
- **batch** (`Batch`) – The batch to migrate to (or the current batch).

### 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.

**Constructor:**

__init__ (`parent=None`)

Create a group.

**Parameters** `parent` (`Group`) – Group to contain this group; its state will be set before this state’s.

**Methods:**

- **set_state()** Apply the OpenGL state change.
- **set_state_recursive()** Set this group and its ancestry.
- **unset_state()** Repeal the OpenGL state change.
- **unset_state_recursive()** Unset this group and its ancestry.

**Methods**

Group.<code>set_state()</code>

Apply the OpenGL state change.

The default implementation does nothing.

Group.<code>set_state_recursive()</code>

Set this group and its ancestry.

Call this method if you are using a group in isolation: the parent groups will be called in top-down order, with this class’s `set` being called last.
Group.

**unset_state**()  
Repeal the OpenGL state change.  
The default implementation does nothing.

Group.

**unset_state_recursive**()  
Unset this group and its ancestry.  
The inverse of set_state_recursive.

---

**NullGroup Class**

**class NullGroup**(parent=None)  
The default group class used when None is given to a batch.  
This implementation has no effect.

**Constructor:**

**init**(parent=None)  
Create a group.

**Parameters** parent (Group) – Group to contain this group; its state will be set before this state’s.

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>set_state</strong>()</td>
<td>Apply the OpenGL state change.</td>
</tr>
<tr>
<td><strong>set_state_recursive</strong>()</td>
<td>Set this group and its ancestry.</td>
</tr>
<tr>
<td><strong>unset_state</strong>()</td>
<td>Repeal the OpenGL state change.</td>
</tr>
<tr>
<td><strong>unset_state_recursive</strong>()</td>
<td>Unset this group and its ancestry.</td>
</tr>
</tbody>
</table>

**Inherited members**

**Methods**

**NullGroup.set_state**()  
Apply the OpenGL state change.  
The default implementation does nothing.

**NullGroup.set_state_recursive**()  
Set this group and its ancestry.  
Call this method if you are using a group in isolation: the parent groups will be called in top-down order, with this class’s set being called last.
NullGroup.unset_state()
    Repeal the OpenGL state change.
    The default implementation does nothing.

NullGroup.unset_state_recursive()
    Unset this group and its ancestry.
    The inverse of set_state_recursive.

---

**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.

Constructor:
    __init__ (order, parent=None)
        Create an ordered group.

Parameters
    • order (int) – Order of this group.
    • parent (Group) – Parent of this group.

Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_state()</td>
<td>Apply the OpenGL state change.</td>
</tr>
<tr>
<td>set_state_recursive()</td>
<td>Set this group and its ancestry.</td>
</tr>
<tr>
<td>unset_state()</td>
<td>Repeal the OpenGL state change.</td>
</tr>
<tr>
<td>unset_state_recursive()</td>
<td>Unset this group and its ancestry.</td>
</tr>
</tbody>
</table>

---

Inherited members

Methods

OrderedGroup.set_state()
    Apply the OpenGL state change.
    The default implementation does nothing.

OrderedGroup.set_state_recursive()
    Set this group and its ancestry.
Call this method if you are using a group in isolation: the parent groups will be called in top-down order, with this class’s set being called last.

OrderedGroup.unset_state()

Repeal the OpenGL state change.

The default implementation does nothing.

OrderedGroup.unset_state_recursive()

Unset this group and its ancestry.

The inverse of set_state_recursive.

---

### 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.

Constructor:

__init__ (texture, parent=None)

Create a texture group.

Parameters

- **texture** (Texture) – Texture to bind.
- **parent** (Group) – Parent group.

Methods:

- set_state()
- set_state_recursive() Set this group and its ancestry.
- unset_state()
- unset_state_recursive() Unset this group and its ancestry.

---

Methods

TextureGroup.set_state()
TextureGroup.unset_state()

Inherited members
 Methods

TextureGroup.set_state_recursive()
Set this group and its ancestry.

Call this method if you are using a group in isolation: the parent groups will be called in top-down order, with this class’s set being called last.

TextureGroup.unset_state_recursive()
Unset this group and its ancestry.

The inverse of set_state_recursive.

 Functions

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(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(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.

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 of int) – 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 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

`compat_platform = `‘linux2’`
str(object=’’)-> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

`null_group = <pyglet.graphics.NullGroup object>`
The default group.

Type Group

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:

```python
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:

```python
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

<table>
<thead>
<tr>
<th>atlas</th>
<th>Group multiple small images into larger textures.</th>
</tr>
</thead>
<tbody>
<tr>
<td>codecs</td>
<td>Collection of image encoders and decoders.</td>
</tr>
</tbody>
</table>

### `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:

```python
# 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

```
Allocator       Rectangular area allocation algorithm.
TextureAtlas   Collection of images within a texture.
TextureBin     Collection of texture atlases.
```
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 decreas-
    ing height order.

    Constructor:

    __init__ (width, height)
    Create an Allocator of the given size.

    Parameters
    • width (int) – Width of the allocation region.
    • height (int) – Height of the allocation region.

    Methods:

    alloc(width, height) Get a free area in the allocator of the given size.
    get_fragmentation() Get the fraction of area that’s unlikely to ever be used, based on current allocation behaviour.
    get_usage() Get the fraction of area already allocated.

Methods

Allocator.alloc (width, height)
Get a free area in the allocator of the given size.

    After calling alloc, the requested area will no longer be used. If there is not enough room to fit the given area
    AllocatorException is raised.

    Parameters
    • width (int) – Width of the area to allocate.
    • height (int) – Height of the area to allocate.

    Return type  int, int

    Returns  The X and Y coordinates of the bottom-left corner of the allocated region.

Allocator.get_fragmentation ()
Get the fraction of area that’s unlikely to ever be used, based on current allocation behaviour.

This method is useful for debugging and profiling only.

    Return type  float
Allocator.get_usage()
Get the fraction of area already allocated.

This method is useful for debugging and profiling only.

Return type  float

TextureAtlas Class

class TextureAtlas (width=256, height=256)
Collection of images within a texture.

Constructor:

__init__(width=256, height=256)
Create a texture atlas of the given size.

Parameters

• width (int) – Width of the underlying texture.
• height (int) – Height of the underlying texture.

Methods:

add(img)  Add an image to the atlas.

Methods

TextureAtlas.add(img)
Add an image to the atlas.

This method will fail if the given image cannot be transferred directly to a texture (for example, if it is another texture). ImageData is the usual image type for this method.

AllocatorException will be raised if there is no room in the atlas for the image.

Parameters  img (AbstractImage) – The image to add.

Return type  TextureRegion

Returns  The region of the atlas containing the newly added image.
TextureBin Class

class TextureBin (texture_width=256, texture_height=256)

Collection of texture atlases.

TextureBin maintains a collection of texture atlases, and creates new ones as necessary to accommodate images added to the bin.

Constructor:

__init__ (texture_width=256, texture_height=256)
Create a texture bin for holding atlases of the given size.

Parameters

• texture_width (int) – Width of texture atlases to create.
• texture_height (int) – Height of texture atlases to create.

Methods:

add(img)
Add an image into this texture bin.

This method calls TextureAtlas.add for the first atlas that has room for the image.

AllocatorException is raised if the image exceeds the dimensions of texture_width and texture_height.

Parameters

img (AbstractImage) – The image to add.

Return type

TextureRegion

Returns

The region of an atlas containing the newly added image.

Exceptions

AllocatorException

The allocator does not have sufficient free space for the requested image size.
**AllocatorException**

Exception defined in `pyglet.image.atlas`

except `AllocatorException`

The allocator does not have sufficient free space for the requested image size.

---

**Notes**

- `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:

```python
def get_decoders():
    # Return a list of ImageDecoder instances or []
    return []

def get_encoders():
    # Return a list of ImageEncoder instances or []
    return []
```

<table>
<thead>
<tr>
<th>Decoder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bmp</code></td>
<td>Decoder for BMP files.</td>
</tr>
<tr>
<td><code>dds</code></td>
<td>DDS texture loader.</td>
</tr>
<tr>
<td><code>gif</code></td>
<td>Read GIF control data.</td>
</tr>
<tr>
<td><code>png</code></td>
<td>Encoder and decoder for PNG files, using PyPNG (png.py).</td>
</tr>
<tr>
<td><code>s3tc</code></td>
<td>Software decoder for S3TC compressed texture (i.e., DDS).</td>
</tr>
</tbody>
</table>

---

**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.


**pyglet.image.codecs.gif** Read GIF control data.
pyglet Documentation, Release 1.2.4

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).


```
ImageDecoder
ImageEncoder
```

### Classes

#### ImageDecoder Class

class ImageDecoder

Methods:

- `decode(file, filename)`     Decode the given file object and return an instance of `Image`.  
- `decode_animation(file, filename)`     Decode the given file object and return an instance of `Animation`.  
- `get_animation_file_extensions()`     Return a list of accepted file extensions, e.g.
- `get_file_extensions()`     Return a list of accepted file extensions, e.g.

#### Methods

**ImageDecoder**.`decode (file, filename)`   
Decode the given file object and return an instance of `Image`.  Throws `ImageDecodeException` if there is an error.  `filename` can be a file type hint.

**ImageDecoder**.`decode_animation (file, filename)`   
Decode the given file object and return an instance of `Animation`.  Throws `ImageDecodeException` if there is an error.  `filename` can be a file type hint.

**ImageDecoder**.`get_animation_file_extensions ()`   
Return a list of accepted file extensions, e.g. `['.gif', '.flc']` Lower-case only.

**ImageDecoder**.`get_file_extensions ()`   
Return a list of accepted file extensions, e.g. `['.png', '.bmp']` Lower-case only.
ImageEncoder Class

class ImageEncoder

Methods:

- **encode**(image, file, filename[, options])  
  Encode the given image to the given file. filename provides a hint to the file format desired. options are encoder-specific, and unknown options should be ignored or issue warnings.

- **get_file_extensions**()  
  Return a list of accepted file extensions, e.g. 

Exceptions

- **ImageDecodeException**  
  Exception defined in `pyglet.image.codecs`

- **ImageEncodeException**  
  Exception defined in `pyglet.image.codecs`
Exception defined in `pyglet.image.codecs`

```python
exception ImageEncodeException
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_decoders(module)</code></td>
<td>Add a decoder module. The module must define <code>get_decoders</code>. Once added, the appropriate decoders defined in the codec will be returned by pyglet.image.codecs.get_decoders.</td>
</tr>
<tr>
<td><code>add_default_image_codecs()</code></td>
<td></td>
</tr>
<tr>
<td><code>add_encoders(module)</code></td>
<td>Add an encoder module. The module must define <code>get_encoders</code>. Once added, the appropriate encoders defined in the codec will be returned by pyglet.image.codecs.get_encoders.</td>
</tr>
<tr>
<td><code>get_animation_decoders([filename])</code></td>
<td>Get an ordered list of decoders to attempt.</td>
</tr>
<tr>
<td><code>get_decoders([filename])</code></td>
<td>Get an ordered list of decoders to attempt.</td>
</tr>
<tr>
<td><code>get_encoders([filename])</code></td>
<td>Get an ordered list of encoders to attempt.</td>
</tr>
</tbody>
</table>

**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_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 = 'linux2'`

`str(object='') -> string`

Return a nice string representation of the object. If the argument is a string, the return value is the same object.
Defined

Notes
• gdkpixbuf2
• os
• pil

Classes

AbstractImage
Abstract class representing an image.

AbstractImageSequence
Abstract sequence of images.

Animation
Sequence of images with timing information.

AnimationFrame
A single frame of an animation.

BufferImage
An abstract framebuffer.

BufferImageMask
A single bit of the stencil buffer.

BufferManager
Manages the set of framebuffers for a context.

CheckerImagePattern
Create an image with a tileable checker image.

ColorBufferImage
A color framebuffer.

CompressedImageData
Image representing some compressed data suitable for direct uploading to driver.

DepthBufferImage
The depth buffer.

DepthTexture
A texture with depth samples (typically 24-bit).

ImageData
An image represented as a string of unsigned bytes.

ImageDataRegion

ImageGrid
An imaginary grid placed over an image allowing easy access to regular regions of that image.

ImagePattern
Abstract image creation class.

SolidColorImagePattern
Creates an image filled with a solid color.

Texture
An image loaded into video memory that can be efficiently drawn to the framebuffer.

Texture3D
A texture with more than one image slice.

TextureGrid
A texture containing a regular grid of texture regions.

TextureRegion
A rectangular region of a texture, presented as if it were a separate texture.

TextureSequence
Interface for a sequence of textures.

TileableTexture
A texture that can be tiled efficiently.

UniformTextureSequence
Interface for a sequence of textures, each with the same dimensions.

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

**Constructor:**

```python
__init__(width, height)
```

**Methods:**

- `blit(x, y[, z])` Draw this image to the active framebuffers.
- `blit_into(source, x, y, z)` Draw `source` on this image.
- `blit_to_texture(target, level, x, y[, z])` Draw this image on the currently bound texture at `target`.
- `get_image_data()` Get an ImageData view of this image.
- `get_mipmapped_texture()` Retrieve a Texture instance with all mipmap levels filled in.
- `get_region(x, y, width, height)` Retrieve a rectangular region of this image.
- `get_texture([rectangle, force_rectangle])` A Texture view of this image.
- `save([filename, file, encoder])` Save this image to a file.

**Attributes:**

- `anchor_x`
- `anchor_y`
- `image_data` An ImageData view of this image.
- `mipmapped_texture` A Texture view of this image.
- `texture` Get a Texture view of this image.

**Methods**

AbstractImage.

- `.blit(x, y, z=0)` Draw this image to the active framebuffers.

  The image will be drawn with the lower-left corner at `(x - anchor_x, y - anchor_y, z)`.

AbstractImage.

- `.blit_into(source, x, y, z)` Draw `source` on this image.

  `source` will be copied into this image such that its anchor point is aligned with the `x` and `y` parameters. If this image is a 3D texture, the `z` coordinate gives the image slice to copy into.

  Note that if `source` is larger than this image (or the positioning would cause the copy to go out of bounds) then you must pass a region of `source` to this method, typically using `get_region()`.

AbstractImage.

- `.blit_to_texture(target, level, x, y, z=0)` Draw this image on the currently bound texture at `target`.

  This image is copied into the texture such that this image’s anchor point is aligned with the given `x` and `y` coordinates of the destination texture. If the currently bound texture is a 3D texture, the `z` coordinate gives the image slice to blit into.

AbstractImage.

- `.get_image_data()` Get an ImageData view of this image.

  Changes to the returned instance may or may not be reflected in this image.

  **Return type** ImageData

**Note:** Since pyglet 1.1
AbstractImage.get_mipmapped_texture()
Retrieve a Texture instance with all mipmap levels filled in.
Requires that image dimensions be powers of 2.

Return type: Texture

Note: Since pyglet 1.1

AbstractImage.get_region(x, y, width, height)
Retrieve a rectangular region of this image.

Parameters
- x (int) – Left edge of region.
- y (int) – Bottom edge of region.
- width (int) – Width of region.
- height (int) – Height of region.

Return type: AbstractImage

AbstractImage.get_texture(rectangle=False, force_rectangle=False)
A Texture view of this image.

By default, textures are created with dimensions that are powers of two. Smaller images will return a TextureRegion that covers just the image portion of the larger texture. This restriction is required on older video cards, and for compressed textures, or where texture repeat modes will be used, or where mipmapping is desired.

If the rectangle parameter is True, this restriction is ignored and a texture the size of the image may be created if the driver supports the GL_ARB_texture_rectangle or GL_NV_texture_rectangle extensions. If the extensions are not present, the image already is a texture, or the image has power 2 dimensions, the rectangle parameter is ignored.

Examine Texture.target to determine if the returned texture is a rectangle (GL_TEXTURE_RECTANGLE_ARB or GL_TEXTURE_RECTANGLE_NV) or not (GL_TEXTURE_2D).

If the force_rectangle parameter is True, one of these extensions must be present, and the returned texture always has target GL_TEXTURE_RECTANGLE_ARB or GL_TEXTURE_RECTANGLE_NV.

Changes to the returned instance may or may not be reflected in this image.

Parameters
- rectangle (bool) – True if the texture can be created as a rectangle.
- force_rectangle (bool) – True if the texture must be created as a rectangle. Since: pyglet 1.1.4.

Return type: Texture

Note: Since pyglet 1.1

AbstractImage.save(filename=None, file=None, encoder=None)
Save this image to a file.

Parameters
- filename (str) – Used to set the image file format, and to open the output file if file is unspecified.
• **file** (file-like object or None) – File to write image data to.

• **encoder** (ImageEncoder or None) – If unspecified, all encoders matching the filename extension are tried. If all fail, the exception from the first one attempted is raised.

**Attributes**

**AbstractImage**.anchor_x = 0

**AbstractImage**.anchor_y = 0

**AbstractImage**.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

---

**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__).

**Methods**:

- **get_animation**(period[, loop])  Create an animation over this image sequence for the given constant framerate.
- **get_texture_sequence`()  Get a TextureSequence.
Attributes:

| texture_sequence | Access this image sequence as a texture sequence. |

Methods

AbstractImageSequence.get_animation(period, loop=True)

Create an animation over this image sequence for the given constant framerate.

:Parameters

- **period** [float] Number of seconds to display each frame.
- **loop** [bool] If True, the animation will loop continuously.

:return: Animation

Note: Since pyglet 1.1

AbstractImageSequence.get_texture_sequence()

Get a TextureSequence.

:rtype: TextureSequence

Note: Since pyglet 1.1

Attributes

AbstractImageSequence.texture_sequence

Access this image sequence as a texture sequence.

Warning: Deprecated. Use get_texture_sequence

Type: TextureSequence

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.

Constructor:
**__init__***(frames)*)
Create an animation directly from a list of frames.

**Parameters** frames *(list of AnimationFrame)* – The frames that make up the animation.

**Methods:**

- **add_to_texture_bin**(bin)
  Add the images of the animation to a TextureBin.

- **from_image_sequence**(sequence, period[, loop])
  Create an animation from a list of images and a constant framerate.

- **get_duration**()
  Get the total duration of the animation in seconds.

- **get_max_height**()
  Get the maximum image frame height.

- **get_max_width**()
  Get the maximum image frame width.

- **get_transform**(flip_x=False, flip_y=False, rotate=0)
  Create a copy of this animation applying a simple transformation.

**Methods**

Animation.**add_to_texture_bin**(bin)
Add the images of the animation to a TextureBin.

The animation frames are modified in-place to refer to the texture bin regions.

**Parameters** bin *(TextureBin)* – Texture bin to upload animation frames into.

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

Animation.**get_duration**()
Get the total duration of the animation in seconds.

**Return type** float

Animation.**get_max_height**()
Get the maximum image frame height.

This method is useful for determining texture space requirements: due to the use of anchor_y the actual required playback area may be larger.

**Return type** int

Animation.**get_max_width**()
Get the maximum image frame width.

This method is useful for determining texture space requirements: due to the use of anchor_x the actual required playback area may be larger.

**Return type** int

Animation.**get_transform**(flip_x=False, flip_y=False, rotate=0)
Create a copy of this animation applying a simple transformation.

The transformation is applied around the image’s anchor point of each frame. The texture data is shared between the original animation and the transformed animation.
Parameters

- **flip_x** *(bool)* – If True, the returned animation will be flipped horizontally.
- **flip_y** *(bool)* – If True, the returned animation will be flipped vertically.
- **rotate** *(int)* – Degrees of clockwise rotation of the returned animation. Only 90-degree increments are supported.

Return type *Animation*

---

**AnimationFrame** class

class **AnimationFrame** *(image, duration)*

A single frame of an animation.

Constructor:

```python
__init__(image, duration)
```

Create an animation frame from an image.

Parameters

- **image** *(AbstractImage)* – The image of this frame.
- **duration** *(float)* – Number of seconds to display the frame, or *None* if it is the last frame in the animation.

---

**BufferImage** class

class **BufferImage** *(x, y, width, height)*

An abstract framebuffer.

Constructor:

```python
__init__(x, y, width, height)
```

Methods:

```python
blit(x, y[, z])
blit_into(source, x, y, z)
blit_to_texture(target, level, x, y[, z])
get_image_data()
```

Draw this image to the active framebuffers.

Draw *source* on this image.

Draw this image on the currently bound texture at *target*.

Continued on next page
Table 2.126 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_mipmapped_texture()</td>
<td>Retrieve a Texture instance with all mipmap levels filled in.</td>
</tr>
<tr>
<td>get_region(x, y, width, height)</td>
<td>A Texture view of this image.</td>
</tr>
<tr>
<td>get_texture([rectangle, force_rectangle])</td>
<td>A Texture view of this image.</td>
</tr>
<tr>
<td>save([filename, file, encoder])</td>
<td>Save this image to a file.</td>
</tr>
</tbody>
</table>

Attributes:

- anchor_x
- anchor_y
- format
- image_data
- mipmapped_texture
- owner
- texture

Methods

BufferImage.get_image_data()  
BufferImage.get_region(x, y, width, height)

Attributes  
BufferImage.format = ''  
The format string used for image data.  
BufferImage.owner = None  

Inherited members

Methods

BufferImage.blit(x, y, z=0)  
Draw this image to the active framebuffers.  
The image will be drawn with the lower-left corner at (x - anchor_x, y - anchor_y, z).

BufferImage.blit_into(source, x, y, z)  
Draw source on this image.  
source will be copied into this image such that its anchor point is aligned with the x and y parameters.  
If this image is a 3D texture, the z coordinate gives the image slice to copy into.  
Note that if source is larger than this image (or the positioning would cause the copy to go out of bounds) then you must pass a region of source to this method, typically using get_region().

BufferImage.blit_to_texture(target, level, x, y, z=0)  
Draw this image on the currently bound texture at target.  
This image is copied into the texture such that this image’s anchor point is aligned with the given x and y coordinates of the destination texture. If the currently bound texture is a 3D texture, the z coordinate gives the image slice to blit into.
BufferImage.get_mipmapped_texture()
Retrieve a Texture instance with all mipmap levels filled in.

Requires that image dimensions be powers of 2.

Return type: Texture

Note: Since pyglet 1.1

BufferImage.get_texture(rectangle=False, force_rectangle=False)
A Texture view of this image.

By default, textures are created with dimensions that are powers of two. Smaller images will return a TextureRegion that covers just the image portion of the larger texture. This restriction is required on older video cards, and for compressed textures, or where texture repeat modes will be used, or where mipmapping is desired.

If the rectangle parameter is True, this restriction is ignored and a texture the size of the image may be created if the driver supports the GL_ARB_texture_rectangle or GL_NV_texture_rectangle extensions. If the extensions are not present, the image already is a texture, or the image has power 2 dimensions, the rectangle parameter is ignored.

Examine Texture.target to determine if the returned texture is a rectangle (GL_TEXTURE_RECTANGLE_ARB or GL_TEXTURE_RECTANGLE_NV) or not (GL_TEXTURE_2D).

If the force_rectangle parameter is True, one of these extensions must be present, and the returned texture always has target GL_TEXTURE_RECTANGLE_ARB or GL_TEXTURE_RECTANGLE_NV.

Changes to the returned instance may or may not be reflected in this image.

Parameters
- rectangle (bool) – True if the texture can be created as a rectangle.
- force_rectangle (bool) – True if the texture must be created as a rectangle.

Since: pyglet 1.1.4.

Return type: Texture

Note: Since pyglet 1.1

BufferImage.save(filename=None, file=None, encoder=None)
Save this image to a file.

Parameters
- filename (str) – Used to set the image file format, and to open the output file if file is unspecified.
- file (file-like object or None) – File to write image data to.
- encoder (ImageEncoder or None) – If unspecified, all encoders matching the filename extension are tried. If all fail, the exception from the first one attempted is raised.

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.mipmapmapped_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_mipmapmed_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

```python
class BufferImageMask(x, y, width, height)
A single bit of the stencil buffer.
```

**Constructor:**

```python
__init__(x, y, width, height)
```

**Methods:**

```python
blit(x, y[, z]) Draw this image to the active framebuffers.
blit_into(source, x, y, z) Draw source on this image.
blit_to_texture(target, level, x, y[, z]) Draw this image on the currently bound texture at target.
get_image_data() Retrieve a Texture instance with all mipmap levels filled in.
get_mipmapmed_texture() Get a Texture view of this image.
get_region(x, y, width, height) Save this image to a file.
```

**Attributes:**

```python
```
anchor_x
anchor_y
format
image_data An ImageData view of this image.
mipmapped_texture A Texture view of this image.
owner
texture Get a Texture view of this image.

Attributes
BufferImageMask.format = ‘L’

Inherited members

Methods

BufferImageMask.blit(x, y, z=0)
Draw this image to the active framebuffers.
The image will be drawn with the lower-left corner at (x - anchor_x, y - anchor_y, z).

BufferImageMask.blit_into(source, x, y, z)
Draw source on this image.
source will be copied into this image such that its anchor point is aligned with the x and y parameters.
If this image is a 3D texture, the z coordinate gives the image slice to copy into.
Note that if source is larger than this image (or the positioning would cause the copy to go out of bounds) then you must pass a region of source to this method, typically using get_region().

BufferImageMask.blit_to_texture(target, level, x, y, z=0)
Draw this image on the currently bound texture at target.
This image is copied into the texture such that this image’s anchor point is aligned with the given x and y coordinates of the destination texture. If the currently bound texture is a 3D texture, the z coordinate gives the image slice to blit into.

BufferImageMask.get_image_data()

BufferImageMask.get_mipmapped_texture()
Retrieve a Texture instance with all mipmap levels filled in.
Requires that image dimensions be powers of 2.

Return type Texture

Note: Since pyglet 1.1

BufferImageMask.get_region(x, y, width, height)

BufferImageMask.get_texture(rectangle=False, force_rectangle=False)
A Texture view of this image.
By default, textures are created with dimensions that are powers of two. Smaller images will return a TextureRegion that covers just the image portion of the larger texture. This restriction is required
on older video cards, and for compressed textures, or where texture repeat modes will be used, or where mipmapping is desired.

If the `rectangle` parameter is True, this restriction is ignored and a texture the size of the image may be created if the driver supports the GL_ARB_texture_rectangle or GL_NV_texture_rectangle extensions. If the extensions are not present, the image already is a texture, or the image has power 2 dimensions, the `rectangle` parameter is ignored.

Examine `Texture.target` to determine if the returned texture is a rectangle (GL_TEXTURE_RECTANGLE_ARB or GL_TEXTURE_RECTANGLE_NV) or not (GL_TEXTURE_2D).

If the `force_rectangle` parameter is True, one of these extensions must be present, and the returned texture always has target GL_TEXTURE_RECTANGLE_ARB or GL_TEXTURE_RECTANGLE_NV.

Changes to the returned instance may or may not be reflected in this image.

**Parameters**

- `rectangle` (bool) – True if the texture can be created as a rectangle.
- `force_rectangle` (bool) – True if the texture must be created as a rectangle.

**Return type** `Texture`

**Note:** Since pyglet 1.1.4.

### BufferImageMask.save

Save this image to a file.

**Parameters**

- `filename` (str) – Used to set the image file format, and to open the output file if `file` is unspecified.
- `file` (file-like object or None) – File to write image data to.
- `encoder` (ImageEncoder or None) – If unspecified, all encoders matching the file-name extension are tried. If all fail, the exception from the first one attempted is raised.

**Attributes**

- `anchor_x` = 0
- `anchor_y` = 0
- `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`

- `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`

```python
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.

**Constructor:**

```
__init__()
```

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_aux_buffer()</code></td>
<td>Get a free auxiliary buffer.</td>
</tr>
<tr>
<td><code>get_buffer_mask()</code></td>
<td>Get a free bitmask buffer.</td>
</tr>
<tr>
<td><code>get_color_buffer()</code></td>
<td>Get the color buffer.</td>
</tr>
<tr>
<td><code>get_depth_buffer()</code></td>
<td>Get the depth buffer.</td>
</tr>
<tr>
<td><code>get_viewport()</code></td>
<td>Get the current OpenGL viewport dimensions.</td>
</tr>
</tbody>
</table>

**Methods**

BufferManager.`get_aux_buffer()`
Get a free auxiliary buffer.

If not aux buffers are available, `ImageException` is raised. Buffers are released when they are garbage collected.

**Return type** `ColorBufferImage`

BufferManager.`get_buffer_mask()`
Get a free bitmask buffer.

A bitmask buffer is a buffer referencing a single bit in the stencil buffer. If no bits are free, `ImageException` is raised. Bits are released when the bitmask buffer is garbage collected.

**Return type** `BufferImageMask`
BufferManager.get_color_buffer()
Get the color buffer.

Return type  ColorBufferImage
BufferManager.get_depth_buffer()
Get the depth buffer.

Return type  DepthBufferImage
BufferManager.get_viewport()
Get the current OpenGL viewport dimensions.

Return type  4-tuple of float.
Returns  Left, top, right and bottom dimensions.

CheckerImagePattern Class

class CheckerImagePattern (color1=(150, 150, 150, 255), color2=(200, 200, 200, 255))
Create an image with a tileable checker image.

Constructor:

__init__ (color1=(150, 150, 150, 255), color2=(200, 200, 200, 255))
Initialise with the given colors.

Parameters

• color1 ((int, int, int, int)) – 4-tuple of ints in range [0,255] giving RGBA components of color to fill with. This color appears in the top-left and bottom-right corners of the image.

• color2 ((int, int, int, int)) – 4-tuple of ints in range [0,255] giving RGBA components of color to fill with. This color appears in the top-right and bottom-left corners of the image.

Methods:

create_image(width, height)

Methods

CheckerImagePattern.create_image (width, height)
**ColorBufferImage Class**

```python
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.

**Constructor:**

```
__init__(x, y, width, height)
```

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blit(x, y[, z])</td>
<td>Draw this image to the active framebuffers.</td>
</tr>
<tr>
<td>blit_into(source, x, y, z)</td>
<td>Draw source on this image.</td>
</tr>
<tr>
<td>blit_to_texture(target, level, x, y, z)</td>
<td>Draw source on this image.</td>
</tr>
<tr>
<td>get_image_data()</td>
<td></td>
</tr>
<tr>
<td>get_mipmapped_texture()</td>
<td>Retrieve a Texture instance with all mipmap levels filled in.</td>
</tr>
<tr>
<td>get_region(x, y, width, height)</td>
<td></td>
</tr>
<tr>
<td>get_texture([rectangle, force_rectangle])</td>
<td></td>
</tr>
<tr>
<td>save([filename, file, encoder])</td>
<td>Save this image to a file.</td>
</tr>
</tbody>
</table>

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor_x</td>
<td></td>
</tr>
<tr>
<td>anchor_y</td>
<td></td>
</tr>
<tr>
<td>format</td>
<td></td>
</tr>
<tr>
<td>image_data</td>
<td>An ImageData view of this image.</td>
</tr>
<tr>
<td>mipmapped_texture</td>
<td>A Texture view of this image.</td>
</tr>
<tr>
<td>owner</td>
<td></td>
</tr>
<tr>
<td>texture</td>
<td>Get a Texture view of this image.</td>
</tr>
</tbody>
</table>

**Methods**

`ColorBufferImage.blit_to_texture(target, level, x, y, z)`

`ColorBufferImage.get_texture(rectangle=False, force_rectangle=False)`

**Attributes**

`ColorBufferImage.format = 'RGBA'`

**Inherited members**

**Methods**

`ColorBufferImage.blit(x, y, z=0)`

Draw this image to the active framebuffers.

The image will be drawn with the lower-left corner at (x - anchor_x, y - anchor_y, z).

`ColorBufferImage.blit_into(source, x, y)`

Draw source on this image.
source will be copied into this image such that its anchor point is aligned with the x and y parameters. If this image is a 3D texture, the z coordinate gives the image slice to copy into.

Note that if source is larger than this image (or the positioning would cause the copy to go out of bounds) then you must pass a region of source to this method, typically using get_region().

ColorBufferImage.get_image_data()

ColorBufferImage.get_mipmapped_texture()

Retrieve a Texture instance with all mipmap levels filled in.

Return type  Texture

Note:  Since pyglet 1.1

ColorBufferImage.get_region(x, y, width, height)

ColorBufferImage.save(filename=None, file=None, encoder=None)

Save this image to a file.

Parameters

• filename (str) – Used to set the image file format, and to open the output file if file is unspecified.

• file (file-like object or None) – File to write image data to.

• encoder (ImageEncoder or None) – If unspecified, all encoders matching the file-name extension are tried. If all fail, the exception from the first one attempted is raised.

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 mipmap levels 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

CompressedImageData Class

class CompressedImageData (width, height, gl_format, data, extension=None, decoder=None)
Image representing some compressed data suitable for direct uploading to driver.

Constructor:
__init__ (width, height, gl_format, data, extension=None, decoder=None)
Construct a CompressedImageData with the given compressed data.

Parameters
• width (int) – Width of image
• height (int) – Height of image
• gl_format (int) – GL constant giving format of compressed data; for example, GL_COMPRESSED_RGBA_S3TC_DXT5_EXT.
• data (sequence) – String or array/list of bytes giving compressed image data.
• extension (str or None) – If specified, gives the name of a GL extension to check for before creating a texture.
• decoder (function(data, width, height) -> AbstractImage) – A function to decode the compressed data, to be used if the required extension is not present.

Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blit(x, y[, z])</td>
<td>Draw this image to the active framebuffers.</td>
</tr>
<tr>
<td>blit_into(source, x, y, z)</td>
<td>Draw source on this image.</td>
</tr>
<tr>
<td>blit_to_texture(target, level, x, y, z)</td>
<td>Draw source on this image.</td>
</tr>
<tr>
<td>get_image_data()</td>
<td>Get an ImageData view of this image.</td>
</tr>
<tr>
<td>get_mipmapped_texture()</td>
<td>Get an ImageData view of this image.</td>
</tr>
<tr>
<td>get_region(x, y, width, height)</td>
<td>Retrieve a rectangular region of this image.</td>
</tr>
<tr>
<td>get_texture([rectangle, force_rectangle])</td>
<td>Set data for a mipmap level.</td>
</tr>
<tr>
<td>save([filename, file, encoder])</td>
<td>Save this image to a file.</td>
</tr>
</tbody>
</table>

Attributes:
CompressedImageData

Methods

CompressedImageData.blit_to_texture(target, level, x, y, z)

CompressedImageData.get_mipmapped_texture()

CompressedImageData.get_texture(rectangle=False, force_rectangle=False)

CompressedImageData.set_mipmap_data(level, data)

Set data for a mipmap level.

Supplied data gives a compressed image for the given mipmap level. The image must be of the correct dimensions for the level (i.e., width >> level, height >> level); but this is not checked. If any mipmap levels are specified, they are used; otherwise, mipmaps for mipmapped_texture are generated automatically.

Parameters

- level (int) – Level of mipmap image to set.
- data (sequence) – String or array/list of bytes giving compressed image data. Data must be in same format as specified in constructor.

Inherited members

Methods

CompressedImageData.blit(x, y, z=0)

Draw this image to the active framebuffers.

The image will be drawn with the lower-left corner at \((x - \text{anchor}_x, y - \text{anchor}_y, z)\).

CompressedImageData.blit_into(source, x, y, z)

Draw source on this image.

source will be copied into this image such that its anchor point is aligned with the x and y parameters. If this image is a 3D texture, the z coordinate gives the image slice to copy into.

Note that if source is larger than this image (or the positioning would cause the copy to go out of bounds) then you must pass a region of source to this method, typically using get_region().

CompressedImageData.get_image_data()

Get an ImageData view of this image.

Changes to the returned instance may or may not be reflected in this image.

Return type ImageData

Note: Since pyglet 1.1
CompressedImageData.get_region(x, y, width, height)
Retrieve a rectangular region of this image.

Parameters
• x (int) – Left edge of region.
• y (int) – Bottom edge of region.
• width (int) – Width of region.
• height (int) – Height of region.

Return type AbstractImage

CompressedImageData.save(filename=None, file=None, encoder=None)
Save this image to a file.

Parameters
• filename (str) – Used to set the image file format, and to open the output file if 
  file is unspecified.
• file (file-like object or None) – File to write image data to.
• encoder (ImageEncoder or None) – If unspecified, all encoders matching the file-
  name extension are tried. If all fail, the exception from the first one attempted is 
  raised.

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.textures
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.

**Constructor:**  
```python
__init__(x, y, width, height)
```

**Methods:**

- `blit(x, y[, z])` Draw this image to the active framebuffers.
- `blit_into(source, x, y, z)` Draw `source` on this image.
- `blit_to_texture(target, level, x, y, z)`
- `get_image_data()` Retrieve a `Texture` instance with all mipmap levels filled in.
- `get_mipmapped_texture()`
- `get_region(x, y, width, height)`
- `get_texture([rectangle, force_rectangle])`
- `save([filename, file, encoder])` Save this image to a file.

**Attributes:**

- `anchor_x`
- `anchor_y`
- `format`
- `image_data` An `ImageData` view of this image.
- `mipmapped_texture` A `Texture` view of this image.
- `owner`
- `texture` Get a `Texture` view of this image.

**Methods**

- `DepthBufferImage.blit_to_texture(target, level, x, y, z)`
- `DepthBufferImage.get_texture(rectangle=False, force_rectangle=False)`

**Attributes**

- `DepthBufferImage.format = 'L'`

**Inherited members**
Methods

DepthBufferImage.blit(x, y, z=0)
Draw this image to the active framebuffers.

The image will be drawn with the lower-left corner at \((x - anchor_x, y - anchor_y, z)\).

DepthBufferImage.blit_into(source, x, y, z)
Draw source on this image.

source will be copied into this image such that its anchor point is aligned with the x and y parameters. If this image is a 3D texture, the z coordinate gives the image slice to copy into.

Note that if source is larger than this image (or the positioning would cause the copy to go out of bounds) then you must pass a region of source to this method, typically using get_region()().

DepthBufferImage.get_image_data()

DepthBufferImage.get_mipmapped_texture()
Retrieve a Texture instance with all mipmap levels filled in.

Requires that image dimensions be powers of 2.

Return type: Texture

Note: Since pyglet 1.1

DepthBufferImage.get_region(x, y, width, height)

DepthBufferImage.save(filename=None, file=None, encoder=None)
Save this image to a file.

Parameters

- filename (str) – Used to set the image file format, and to open the output file if file is unspecified.
- file (file-like object or None) – File to write image data to.
- encoder (ImageEncoder or None) – If unspecified, all encoders matching the file-name extension are tried. If all fail, the exception from the first one attempted is raised.

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`

```python
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`

```
pyglet.image.AbstractImage       pyglet.image.Texture       pyglet.image.DepthTexture
```

**DepthTexture** Class

```python
class DepthTexture(width, height, target, id)
```

A texture with depth samples (typically 24-bit).

**Constructor:**

```python
__init__(width, height, target, id)
```

**Methods:**

- `blit(x, y[, z, width, height])`
- `blit_into(source, x, y, z)`
- `blit_to_texture(target, level, x, y[, z])` Draw this image on the currently bound texture at `target`.
- `create(width, height[, internalformat, ...])` Create an empty Texture.
- `create_for_size(target, min_width, min_height)` Create a Texture with dimensions at least `min_width`, `min_height`.
- `delete()` Delete the texture from video memory.
- `get_image_data([z])` Get the image data of this texture.
- `get_mipmapped_texture()` Retrieve a `Texture` instance with all mipmap levels filled in.
- `get_region(x, y, width, height)`
- `get_texture([rectangle, force_rectangle])`
- `get_transform([flip_x, flip_y, rotate])` Create a copy of this image applying a simple transformation.
- `save([filename, file, encoder])` Save this image to a file.

**Attributes:**

- `anchor_x`  
- `anchor_y`  
- `image_data` An ImageData view of this texture. Continued on next page
Table 2.139 – continued from previous page

<table>
<thead>
<tr>
<th>images</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>mipmapped_texture</td>
<td>A Texture view of this image.</td>
</tr>
<tr>
<td>tex_coords</td>
<td>tex_coords_order</td>
</tr>
<tr>
<td>texture</td>
<td>Get a Texture view of this image.</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

Methods

DepthTexture.blit_into(source, x, y, z)

Inherited members

Methods

DepthTexture.blit(x, y, z=0, width=None, height=None)

DepthTexture.blit_to_texture(target, level, x, y, z=0)

Draw this image on the currently bound texture at target.

This image is copied into the texture such that this image’s anchor point is aligned with the given x and y coordinates of the destination texture. If the currently bound texture is a 3D texture, the z coordinate gives the image slice to blit into.

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

DepthTexture.delete()
Delete the texture from video memory.

Warning: Deprecated. Textures are automatically released during object finalization.

DepthTexture.get_image_data(z=0)
Get the image data of this texture.

Changes to the returned instance will not be reflected in this texture.

Parameters z (int) – For 3D textures, the image slice to retrieve.

Return type ImageData

DepthTexture.get_mipmapped_texture()
Retrieve a Texture instance with all mipmap levels filled in.

Requires that image dimensions be powers of 2.

Return type Texture

Note: Since pyglet 1.1

DepthTexture.get_region(x, y, width, height)
DepthTexture.get_texture(rectangle=False, force_rectangle=False)
DepthTexture.get_transform(flip_x=False, flip_y=False, rotate=0)
Create a copy of this image applying a simple transformation.
The transformation is applied to the texture coordinates only; `get_image_data` will return the un-transformed data. The transformation is applied around the anchor point.

**Parameters**

- `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)` – Degrees of clockwise rotation of the returned image. Only 90-degree increments are supported.

**Return type** `TextureRegion`

DepthTexture.**save** *(filename=None, file=None, encoder=None)*

Save this image to a file.

**Parameters**

- `filename (str)` – Used to set the image file format, and to open the output file if `file` is unspecified.
- `file (file-like object or None)` – File to write image data to.
- `encoder (ImageEncoder or None)` – If unspecified, all encoders matching the filename extension are tried. If all fail, the exception from the first one attempted is raised.

**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.

**Warning:** Deprecated. Use `get_image_data`.

**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, 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**

```python
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).

**Constructor:**

```python
__init__(width, height, format, data, pitch=None)
```

Initialise image data.

**Parameters**

- **width** (`int`) – Width of image data
- **height** (`int`) – Height of image data
- **format** (`str`) – A valid format string, such as ‘RGB’, ‘RGBA’, ‘ARGB’, etc.
- **data** (`sequence`) – String or array/list of bytes giving the decoded data.
- **pitch** (`int or None`) – If specified, the number of bytes per row. Negative values indicate a top-to-bottom arrangement. Defaults to `width * len(format)`.

**Methods:**

- `blit(x, y[, z, width, height])`
- `blit_into(source, x, y, z)`

Draw `source` on this image.
Table 2.140 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>blit_to_texture(target, level, x, y, z[, ...])</code></td>
<td>Draw this image to the currently bound texture at target.</td>
</tr>
<tr>
<td><code>create_texture(cls[, rectangle, force_rectangle])</code></td>
<td>Create a texture containing this image.</td>
</tr>
<tr>
<td><code>get_data(format, pitch)</code></td>
<td>Get the byte data of the image.</td>
</tr>
<tr>
<td><code>get_image_data()</code></td>
<td></td>
</tr>
<tr>
<td><code>get_mipmapped_texture()</code></td>
<td>Return a Texture with mipmaps.</td>
</tr>
<tr>
<td><code>get_region(x, y, width, height)</code></td>
<td>Retrieve a rectangular region of this image data.</td>
</tr>
<tr>
<td><code>save([filename, file, encoder])</code></td>
<td>Save this image to a file.</td>
</tr>
<tr>
<td><code>set_data(format, pitch, data)</code></td>
<td>Set the byte data of the image.</td>
</tr>
<tr>
<td><code>set_mipmap_image(level, image)</code></td>
<td>Set a mipmap image for a particular level.</td>
</tr>
</tbody>
</table>

**Attributes:**

- `anchor_x`
- `anchor_y`
- `data` - The byte data of the image.
- `format` - Format string of the data.
- `image_data` - An `ImageData` view of this image.
- `mipmapped_texture` - A Texture view of this image.
- `texture` - Get a Texture view of this image.

**Methods**

**ImageData**. `blit(x, y, z=0, width=None, height=None)`  
This image’s anchor point will be aligned to the given x and y coordinates. If the currently bound texture is a 3D texture, the z parameter gives the image slice to blit into.

If `internalformat` is specified, `glTexImage` is used to initialise the texture; otherwise, `glTexSubImage` is used to update a region.

**ImageData**. `create_texture(cls[, rectangle=False, force_rectangle=False])`  
Create a texture containing this image.

If the image’s dimensions are not powers of 2, a TextureRegion of a larger Texture will be returned that matches the dimensions of this image.

**Parameters**

- `cls (class (subclass of Texture))` – Class to construct.
- `rectangle (bool)` – True if a rectangle can be created; see `AbstractImage.get_texture`.  
  Since: pyglet 1.1
- `force_rectangle (bool)` – True if a rectangle must be created; see `AbstractImage.get_texture`. Since: pyglet 1.1.4

**Return type** cls or cls.region_class

**ImageData**. `get_data(format, pitch)`  
Get the byte data of the image.
Parameters

- **format** *(str)* – Format string of the return data.
- **pitch** *(int)* – Number of bytes per row. Negative values indicate a top-to-bottom arrangement.

**Note:** Since pyglet 1.1

Return type  sequence of bytes, or str

**ImageData.get_image_data()**

**ImageData.get_mipmapped_texture()**

Return a Texture with mipmaps.

If **set_mipmap_image** has been called with at least one image, the set of images defined will be used. Otherwise, mipmaps will be automatically generated.

The texture dimensions must be powers of 2 to use mipmaps.

Return type  **Texture**

**Note:** Since pyglet 1.1

**ImageData.get_region(x, y, width, height)**

Retrieve a rectangular region of this image data.

Parameters

- **x** *(int)* – Left edge of region.
- **y** *(int)* – Bottom edge of region.
- **width** *(int)* – Width of region.
- **height** *(int)* – Height of region.

Return type  **ImageDataRegion**

**ImageData.get_texture**(rectangle=False, force_rectangle=False)

**ImageData.set_data**(format, pitch, data)

Set the byte data of the image.

Parameters

- **format** *(str)* – Format string of the return data.
- **pitch** *(int)* – Number of bytes per row. Negative values indicate a top-to-bottom arrangement.
- **data** *(str or sequence of bytes)* – Image data.

**Note:** Since pyglet 1.1

**ImageData.set_mipmap_image**(level, image)

Set a mipmap image for a particular level.

The mipmap image will be applied to textures obtained via **get_mipmapped_texture**.

Parameters
• **level** (*int*) – Mipmap level to set image at, must be >= 1.

• **image** (*AbstractImage*) – Image to set. Must have correct dimensions for that mipmap level (i.e., width >> level, height >> level)

**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**

**Methods**

**ImageData.blit_into** (*source*, *x*, *y*, *z*)

Draw *source* on this image.

*source* will be copied into this image such that its anchor point is aligned with the *x* and *y* parameters. If this image is a 3D texture, the *z* coordinate gives the image slice to copy into.

Note that if *source* is larger than this image (or the positioning would cause the copy to go out of bounds) then you must pass a region of *source* to this method, typically using `get_region()`.

**ImageData.save** (*filename=None*, *file=None*, *encoder=None*)

Save this image to a file.

**Parameters**

• **filename** (*str*) – Used to set the image file format, and to open the output file if *file* is unspecified.

• **file** (*file-like object or None*) – File to write image data to.

• **encoder** (*ImageEncoder or None*) – If unspecified, all encoders matching the filename extension are tried. If all fail, the exception from the first one attempted is raised.

**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`
### Methods

ImageDataRegion\[.\texttt{get\_data}(\text{format, pitch})\]

ImageDataRegion\[.\texttt{get\_region}(x, y, width, height)\]

### Attributes

ImageDataRegion\[.\texttt{data}\]

### Inherited members

#### Methods

ImageDataRegion\[.\texttt{blit}(x, y, z=0, width=\text{None}, height=\text{None})\]

ImageDataRegion\[.\texttt{blit\_into}(\text{source, x, y, z})\]

Draw \text{source} on this image.

\text{source} will be copied into this image such that its anchor point is aligned with the \text{x} and \text{y} parameters. If this image is a 3D texture, the \text{z} coordinate gives the image slice to copy into.

Note that if \text{source} is larger than this image (or the positioning would cause the copy to go out of bounds) then you must pass a region of \text{source} to this method, typically using get_region().

ImageDataRegion\[.\texttt{blit\_to\_texture}(\text{target, level, x, y, z, internalformat=\text{None}})\]

Draw this image to the currently bound texture at \text{target}.

This image’s anchor point will be aligned to the given \text{x} and \text{y} coordinates. If the currently bound texture is a 3D texture, the \text{z} parameter gives the image slice to blit into.

If \text{internalformat} is specified, glTexImage is used to initialise the texture; otherwise, glTexSubImage is used to update a region.

ImageDataRegion\[.\texttt{create\_texture}(\text{cls, rectangle=\text{False}, force\_rectangle=\text{False}})\]

Create a texture containing this image.

If the image’s dimensions are not powers of 2, a TextureRegion of a larger Texture will be returned that matches the dimensions of this image.

#### Parameters

- \text{cls (class (subclass of Texture))} – Class to construct.
- \text{rectangle (bool)} – True if a rectangle can be created; see AbstractImage.get_texture. Since: pyglet 1.1
- \text{force\_rectangle (bool)} – True if a rectangle must be created; see AbstractImage.get_texture. Since: pyglet 1.1.4

#### Return type

\text{cls or cls.region\_class}
ImageDataRegion.get_image_data()

Return a Texture with mipmaps.

If set_mipmap_image has been called with at least one image, the set of images defined will be used. Otherwise, mipmaps will be automatically generated.

The texture dimensions must be powers of 2 to use mipmaps.

**Return type** Texture

**Note:** Since pyglet 1.1

ImageDataRegion.get_texture(rectangle=False, force_rectangle=False)

ImageDataRegion.save(filename=None, file=None, encoder=None)

Save this image to a file.

**Parameters**

- **filename** (str) – Used to set the image file format, and to open the output file if file is unspecified.
- **file** (file-like object or None) – File to write image data to.
- **encoder** (ImageEncoder or None) – If unspecified, all encoders matching the filename extension are tried. If all fail, the exception from the first one attempted is raised.

ImageDataRegion.set_data(format, pitch, data)

Set the byte data of the image.

**Parameters**

- **format** (str) – Format string of the return data.
- **pitch** (int) – Number of bytes per row. Negative values indicate a top-to-bottom arrangement.
- **data** (str or sequence of bytes) – Image data.

**Note:** Since pyglet 1.1

ImageDataRegion.set_mipmap_image(level, image)

Set a mipmap image for a particular level.

The mipmap image will be applied to textures obtained via get_mipmapped_texture.

**Parameters**

- **level** (int) – Mipmap level to set image at, must be >= 1.
- **image** (AbstractImage) – Image to set. Must have correct dimensions for that mipmap level (i.e., width >> level, height >> level)

**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:

```python
image_grid = ImageGrid(...)
texture_grid = image_grid.get_texture_sequence()
```

or as a Texture3D:
Constructor:

```python
__init__(image, rows, columns, item_width=None, item_height=None, row_padding=0, column_padding=0)
```

Construct a grid for the given image.

You can specify parameters for the grid, for example setting the padding between cells. Grids are always aligned to the bottom-left corner of the image.

Parameters

- **image** (*AbstractImage*) – Image over which to construct the grid.
- **rows** (*int*) – Number of rows in the grid.
- **columns** (*int*) – Number of columns in the grid.
- **item_width** (*int*) – Width of each column. If unspecified, is calculated such that the entire image width is used.
- **item_height** (*int*) – Height of each row. If unspecified, is calculated such that the entire image height is used.
- **row_padding** (*int*) – Pixels separating adjacent rows. The padding is only inserted between rows, not at the edges of the grid.
- **column_padding** (*int*) – Pixels separating adjacent columns. The padding is only inserted between columns, not at the edges of the grid.

Methods:

- `blit(x, y[, z])` Draw this image to the active framebuffers.
- `blit_into(source, x, y, z)` Draw `source` on this image.
- `blit_to_texture(target, level, x, y[, z])` Draw this image on the currently bound texture at `target`.
- `get_animation(period[, loop])` Create an animation over this image sequence for the given constant framerate.
- `get_image_data()` Retrieve an `ImageData` view of this image.
- `get_mipmapped_texture()` Retrieve a `Texture` instance with all mipmap levels filled in.
- `get_region(x, y, width, height)` Retrieve a rectangular region of this image.
- `get_texture([rectangle, force_rectangle])` Get a `Texture` view of this image.
- `get_texture_sequence()` Access this image sequence as a texture sequence.
- `save([filename, file, encoder])` Save this image to a file.

Attributes:

- `anchor_x`
- `anchor_y`
- `image_data` An `ImageData` view of this image.
- `mipmapped_texture` A `Texture` view of this image.
- `texture` Get a `Texture` view of this image.
- `texture_sequence` Access this image sequence as a texture sequence.
Methods

ImageGrid.get_image_data()
ImageGrid.get_texture(rectangle=False, force_rectangle=False)
ImageGrid.get_texture_sequence()

Inherited members

Methods

ImageGrid.blit(x, y, z=0)
Draw this image to the active framebuffers.

The image will be drawn with the lower-left corner at \((x - \text{anchor}_x, y - \text{anchor}_y, z)\).

ImageGrid.blit_into(source, x, y, z)
Draw source on this image.

source will be copied into this image such that its anchor point is aligned with the \(x\) and \(y\) parameters. If this image is a 3D texture, the \(z\) coordinate gives the image slice to copy into.

Note that if source is larger than this image (or the positioning would cause the copy to go out of bounds) then you must pass a region of source to this method, typically using get_region().

ImageGrid.blit_to_texture(target, level, x, y, z=0)
Draw this image on the currently bound texture at target.

This image is copied into the texture such that this image’s anchor point is aligned with the given \(x\) and \(y\) coordinates of the destination texture. If the currently bound texture is a 3D texture, the \(z\) coordinate gives the image slice to blit into.

ImageGrid.get_animation(period, loop=True)
Create an animation over this image sequence for the given constant framerate.

:Parameters

- **period** [float] Number of seconds to display each frame.
- **loop** [bool] If True, the animation will loop continuously.

Return type: Animation

Note: Since pyglet 1.1

ImageGrid.get_mipmapped_texture()
Retrieve a Texture instance with all mipmap levels filled in.

Requires that image dimensions be powers of 2.

Return type: Texture

Note: Since pyglet 1.1

ImageGrid.get_region(x, y, width, height)
Retrieve a rectangular region of this image.

Parameters

- **x** (int) – Left edge of region.
• \texttt{y (int)} – Bottom edge of region.
• \texttt{width (int)} – Width of region.
• \texttt{height (int)} – Height of region.

Return type \texttt{AbstractImage}

\texttt{ImageGrid.save(filename=None, file=None, encoder=None)}

Save this image to a file.

Parameters
• \texttt{filename (str)} – Used to set the image file format, and to open the output file if \texttt{file} is unspecified.
• \texttt{file (file-like object or None)} – File to write image data to.
• \texttt{encoder (ImageEncoder or None)} – If unspecified, all encoders matching the file-name extension are tried. If all fail, the exception from the first one attempted is raised.

Attributes

\texttt{ImageGrid.anchor_x = 0}
\texttt{ImageGrid.anchor_y = 0}

\texttt{ImageGrid.image_data}

An \texttt{ImageData} view of this image.

Changes to the returned instance may or may not be reflected in this image. Read-only.

\textbf{Warning:} Deprecated. Use \texttt{get_image_data}.

Type \texttt{ImageData}

\texttt{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.

\textbf{Warning:} Deprecated. Use \texttt{get_mipmapped_texture}.

Type \texttt{Texture}

\texttt{ImageGrid.texture}

Get a \texttt{Texture} view of this image.

Changes to the returned instance may or may not be reflected in this image.

\textbf{Warning:} Deprecated. Use \texttt{get_texture}.

Type \texttt{Texture}

\texttt{ImageGrid.texture_sequence}

Access this image sequence as a texture sequence.

\textbf{Warning:} Deprecated. Use \texttt{get_texture_sequence}.

Type \texttt{TextureSequence}
**ImagePattern Class**

```python
class ImagePattern
    Abstract image creation class.
    Methods:
        create_image(width, height)  
        Create an image of the given size.
```

**Methods**

*ImagePattern*. `create_image` *(width, height)*
Create an image of the given size.

**Parameters**

- `width` *(int)* – Width of image to create
- `height` *(int)* – Height of image to create

**Return type** *AbstractImage*

**SolidColorImagePattern Class**

```python
class SolidColorImagePattern(color=(0, 0, 0, 0))
    Creates an image filled with a solid color.
    Constructor:
        __init__(color=(0, 0, 0, 0))  
        Create a solid image pattern with the given color.
        Parameters color ((int, int, int, int)) – 4-tuple of ints in range [0,255] giving RGBA components of color to fill with.
    Methods:
        create_image(width, height)
```

---

2.1. pyglet  237
Methods

SolidColorImagePattern.create_image(width, height)

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.

Constructor:

__init__ (width, height, target, id)

Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blit(x, y[, z, width, height])</td>
<td>Draw this image on the currently bound texture at target.</td>
</tr>
<tr>
<td>blit_into(source, x, y, z)</td>
<td>Get the image data of this texture.</td>
</tr>
<tr>
<td>blit_to_texture(target, level, x, y[, z])</td>
<td>Retrieve a Texture instance with all mipmap levels filled in.</td>
</tr>
<tr>
<td>create(width, height[, internalformat, ...])</td>
<td>Create a Texture with dimensions at least min_width, min_height.</td>
</tr>
<tr>
<td>create_for_size(target, min_width, min_height)</td>
<td>Create an empty Texture.</td>
</tr>
<tr>
<td>delete()</td>
<td>Delete the texture from video memory.</td>
</tr>
<tr>
<td>get_image_data([z])</td>
<td>Get the image data of this texture.</td>
</tr>
<tr>
<td>get_mipmapped_texture()</td>
<td>Retrieve a Texture instance with all mipmap levels filled in.</td>
</tr>
<tr>
<td>get_region(x, y, width, height)</td>
<td>Create a copy of this image applying a simple transformation.</td>
</tr>
<tr>
<td>get_texture([rectangle, force_rectangle])</td>
<td>Save this image to a file.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor_x</td>
<td></td>
</tr>
<tr>
<td>anchor_y</td>
<td></td>
</tr>
<tr>
<td>image_data</td>
<td>An ImageData view of this texture.</td>
</tr>
<tr>
<td>images</td>
<td></td>
</tr>
</tbody>
</table>
Table 2.149 – continued from previous page

<table>
<thead>
<tr>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>mipmapped_texture</td>
</tr>
<tr>
<td>tex_coords</td>
</tr>
<tr>
<td>tex_coords_order</td>
</tr>
<tr>
<td>texture</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>z</td>
</tr>
</tbody>
</table>

Methods

Texture.blit \((x, y, z=0, width=None, height=None)\)

Texture.blit_into \((source, x, y, z)\)

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*

Texture.delete()
Delete the texture from video memory.

**Warning**: Deprecated. Textures are automatically released during object finalization.

Texture.get_image_data(*z=0*)
Get the image data of this texture.
Changes to the returned instance will not be reflected in this texture.

Parameters
- **z** (*int*) – For 3D textures, the image slice to retrieve.

Return type *ImageData*

Texture.get_region(*x, y, width, height*)

Texture.get_texture(*rectangle=False, force_rectangle=False*)

Texture.get_transform(*flip_x=False, flip_y=False, rotate=0*)
Create a copy of this image applying a simple transformation.
The transformation is applied to the texture coordinates only; *get_image_data* will return the untransformed data. The transformation is applied around the anchor point.

Parameters
- **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*) – Degrees of clockwise rotation of the returned image. Only 90-degree increments are supported.

Return type *TextureRegion*

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, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 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

Methods

Texture.blit_to_texture(target, level, x, y, z=0)
   Draw this image on the currently bound texture at target.
   This image is copied into the texture such that this image’s anchor point is aligned with the given x and y coordinates of the destination texture. If the currently bound texture is a 3D texture, the z coordinate gives the image slice to blit into.

Texture.get_mipmapped_texture()
   Retrieve a Texture instance with all mipmap levels filled in.
   Requires that image dimensions be powers of 2.
   
   Return type: Texture

   Notes: Since pyglet 1.1

Texture.save(filename=None, file=None, encoder=None)
   Save this image to a file.
   
   Parameters:
   - filename (str) – Used to set the image file format, and to open the output file if file is unspecified.
   - file (file-like object or None) – File to write image data to.
   - encoder (ImageEncoder or None) – If unspecified, all encoders matching the file-name extension are tried. If all fail, the exception from the first one attempted is raised.

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

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.

Constructor:

__init__ (width, height, target, id)

Methods:

- blit(x, y, z, width, height)
- blit_into(source, x, y, z)
- blit_to_texture(target, level, x, y, z) Draw this image on the currently bound texture at target.
- create(width, height[, internalformat, ...]) Create an empty Texture.
- create_for_image_grid(grid[, internalformat])
- create_for_images(images[, internalformat])
- create_for_size(target, min_width, min_height) Create a Texture with dimensions at least min_width, min_height.
- delete() Delete the texture from video memory.
- get_animation(period[, loop]) Create an animation over this image sequence for the given constant framerate.
- get_image_data(z) Get the image data of this texture.
- get_mipmapped_texture() Retrieve a Texture instance with all mipmap levels filled in.
- get_region(x, y, width, height)
- get_texture([rectangle, force_rectangle])
- get_texture_sequence() Get the texture sequence for this image.
- get_transform([flip_x, flip_y, rotate]) Create a copy of this image applying a simple transformation.
- save([filename, file, encoder]) Save this image to a file.

Attributes:

- anchor_x
- anchor_y
- image_data An ImageData view of this texture.
- images
- item_height
- item_width

Continued on next page
### Table 2.151 – continued from previous page

<table>
<thead>
<tr>
<th>items</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>mipmapped_texture</td>
<td>A Texture view of this image.</td>
</tr>
<tr>
<td>tex_coords</td>
<td>tex_coords_order</td>
</tr>
<tr>
<td>texture</td>
<td>Get a Texture view of this image.</td>
</tr>
<tr>
<td>texture_sequence</td>
<td>Access this image sequence as a texture sequence.</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

#### 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.blit(x, y, z=0, width=None, height=None)`
- `Texture3D.blit_into(source, x, y, z)`
- `Texture3D.blit_to_texture(target, level, x, y, z=0)`

  Draw this image on the currently bound texture at target.

  This image is copied into the texture such that this image’s anchor point is aligned with the given x and y coordinates of the destination texture. If the currently bound texture is a 3D texture, the z coordinate gives the image slice to blit into.

- `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*

Texture3D.delete *

Delete the texture from video memory.

**Warning:** Deprecated. Textures are automatically released during object finalization.

Texture3D.get_animation *(period, loop=True)*

Create an animation over this image sequence for the given constant framerate.

**Parameters**

• **period** *[float]* Number of seconds to display each frame.

• **loop** *[bool]* If True, the animation will loop continuously.

**Return type** *Animation*
Note: Since pyglet 1.1

Texture3D.get_image_data(z=0)
Get the image data of this texture.
Changes to the returned instance will not be reflected in this texture.

Parameters
z (int) – For 3D textures, the image slice to retrieve.

Return type
ImageData

Texture3D.get_mipmapped_texture()
Retrieve a Texture instance with all mipmap levels filled in.
Requires that image dimensions be powers of 2.

Return type
Texture

Note: Since pyglet 1.1

Texture3D.get_region(x, y, width, height)
Texture3D.get_texture(rectangle=False, force_rectangle=False)
Texture3D.get_texture_sequence()
Texture3D.get_transform(flip_x=False, flip_y=False, rotate=0)
Create a copy of this image applying a simple transformation.
The transformation is applied to the texture coordinates only: get_image_data will return the untransformed data. The transformation is applied around the anchor point.

Parameters

• 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) – Degrees of clockwise rotation of the returned image. Only 90-degree increments are supported.

Return type
TextureRegion

Texture3D.save(filename=None, file=None, encoder=None)
Save this image to a file.

Parameters

• filename (str) – Used to set the image file format, and to open the output file if file is unspecified.

• file (file-like object or None) – File to write image data to.

• encoder (ImageEncoder or None) – If unspecified, all encoders matching the filename extension are tried. If all fail, the exception from the first one attempted is raised.

Attributes

Texture3D.anchor_x = 0
Texture3D.anchor_y = 0
Texture3D\texttt{.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 \texttt{get\_image\_data}. Read-only.

\begin{verbatim}
Warning: Deprecated. Use \texttt{get\_image\_data}.
\end{verbatim}

Type \texttt{ImageData}

Texture3D\texttt{.images = 1}

Texture3D\texttt{.level = 0}

Texture3D\texttt{.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.

\begin{verbatim}
Warning: Deprecated. Use \texttt{get\_mipmapped\_texture}.
\end{verbatim}

Type \texttt{Texture}

Texture3D\texttt{.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\texttt{.tex\_coords\_order = (0, 1, 2, 3)}

Texture3D\texttt{.texture}
Get a \texttt{Texture} view of this image.
Changes to the returned instance may or may not be reflected in this image.

\begin{verbatim}
Warning: Deprecated. Use \texttt{get\_texture}.
\end{verbatim}

Type \texttt{Texture}

Texture3D\texttt{.texture\_sequence}
Access this image sequence as a texture sequence.

\begin{verbatim}
Warning: Deprecated. Use \texttt{get\_texture\_sequence}.
\end{verbatim}

Type \texttt{TextureSequence}

Texture3D\texttt{.x = 0}
Texture3D\texttt{.y = 0}
Texture3D\texttt{.z = 0}

\texttt{TextureGrid} Class

\begin{verbatim}
class TextureGrid(grid)
\end{verbatim}
A texture containing a regular grid of texture regions.
To construct, create an \texttt{ImageGrid} first:
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:

```python
# 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:

```python
# 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:

```python
# 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)]
```

Constructor:

```python
__init__(grid)
```

Methods:

- `blit(x, y, width, height)`
- `blit_into(source, x, y)`
- `blit_to_texture(target, level, x, y)` Draw this image on the currently bound texture at `target`.
- `create(width, height [, internalformat, ...])` Create an empty Texture.
- `create_for_size(target, min_width, min_height)` Create a Texture with dimensions at least `min_width`, `min_height`.
- `delete()` Delete the texture from video memory.
- `get(row, column)`
- `get_animation(period, loop)` Create an animation over this image sequence for the given constant framerate.
- `get_image_data()`
- `get_mipmapped_texture()` Retrieve a Texture instance with all mipmap levels filled in.
- `get_region(x, y, width, height)`
- `get_texture([rectangle, force_rectangle])`
- `get_texture_sequence()`
- `get_transform([flip_x, flip_y, rotate])` Create a copy of this image applying a simple transformation.
- `save([filename, file, encoder])` Save this image to a file.

Attributes:

- `anchor_x`
- `anchor_y`
- `columns`
- `image_data` An ImageData view of this texture.
- `images`
- `item_height`
- `item_width`
Table 2.153 – continued from previous page

<table>
<thead>
<tr>
<th>items</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>mipmapped_texture</td>
<td>A Texture view of this image.</td>
</tr>
<tr>
<td>rows</td>
<td>tex_coords</td>
</tr>
<tr>
<td>tex_coords_order</td>
<td>texture</td>
</tr>
<tr>
<td>texture_sequence</td>
<td>Get a Texture view of this image.</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>z</td>
<td></td>
</tr>
</tbody>
</table>

Methods

TextureGrid.get(row, column)

Attributes

TextureGrid.columns = 1
TextureGrid.item_height = 0
TextureGrid.item_width = 0
TextureGrid.items = ()
TextureGrid.rows = 1

Inherited members

Methods

TextureGrid.blit(x, y, z=0, width=None, height=None)

TextureGrid.blit_into(source, x, y, z)

TextureGrid.blit_to_texture(target, level, x, y, z=0)

Draw this image on the currently bound texture at target.

This image is copied into the texture such that this image’s anchor point is aligned with the given x and y coordinates of the destination texture. If the currently bound texture is a 3D texture, the z coordinate gives the image slice to blit into.

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*

TextureGrid.delete()

Delete the texture from video memory.

**Warning:** Deprecated. Textures are automatically released during object finalization.

TextureGrid.get_animation(period, loop=True)

Create an animation over this image sequence for the given constant framerate.

**:Parameters**

• **period** [float] Number of seconds to display each frame.

• **loop** [bool] If True, the animation will loop continuously.
Return type: `Animation`

**Note:** Since pyglet 1.1

TextureGrid.get_image_data()

TextureGrid.get_mipmapped_texture()

Retrieve a `Texture` instance with all mipmap levels filled in.

Requires that image dimensions be powers of 2.

Return type: `Texture`

**Note:** Since pyglet 1.1

TextureGrid.get_region(x, y, width, height)

TextureGrid.get_texture(rectangle=False, force_rectangle=False)

TextureGrid.get_texture_sequence()

TextureGrid.get_transform(flip_x=False, flip_y=False, rotate=0)

Create a copy of this image applying a simple transformation.

The transformation is applied to the texture coordinates only; `get_image_data` will return the un-transformed data. The transformation is applied around the anchor point.

**Parameters**

- **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)** – Degrees of clockwise rotation of the returned image. Only 90-degree increments are supported.

**Return type:** `TextureRegion`

TextureGrid.save(filename=None, file=None, encoder=None)

Save this image to a file.

**Parameters**

- **filename (str)** – Used to set the image file format, and to open the output file if `file` is unspecified.
- **file (file-like object or None)** – File to write image data to.
- **encoder (ImageEncoder or None)** – If unspecified, all encoders matching the filename extension are tried. If all fail, the exception from the first one attempted is raised.

**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, 0.0, 0.0, 1.0, 0.0, 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.

Constructor:

__init__(x, y, z, width, height, owner)
Methods:

blit(x, y[, z, width, height])
blit_into(source, x, y, z)
blit_to_texture(target, level, x, y[, z])
create(width, height[, internalformat, ...])
create_for_size(target, min_width, min_height)
delete()
get_image_data()
get_mipmapped_texture()
get_region(x, y, width, height)
get_transform([flip_x, flip_y, rotate])
save([filename, file, encoder])

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 Texture view of this image.
x
y
z

Methods

TextureRegion.blit_into(source, x, y, z)
TextureRegion.get_image_data()
TextureRegion.get_region(x, y, width, height)

Inherited members

Methods

TextureRegion.blit(x, y, z=0, width=None, height=None)
TextureRegion.blit_to_texture(target, level, x, y, z=0)

Draw this image on the currently bound texture at target.

This image is copied into the texture such that this image’s anchor point is aligned with the given x and y coordinates of the destination texture. If the currently bound texture is a 3D texture, the z coordinate gives the image slice to blit into.
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

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`

Delete the texture from video memory.
Warning: Deprecated. Textures are automatically released during object finalization.

TextureRegion.get_mipmapped_texture()
Retrieve a Texture instance with all mipmap levels filled in.
Requires that image dimensions be powers of 2.

Return type Texture

Note: Since pyglet 1.1

TextureRegion.get_texture(rectangle=False, force_rectangle=False)
TextureRegion.get_transform(flip_x=False, flip_y=False, rotate=0)
Create a copy of this image applying a simple transformation.
The transformation is applied to the texture coordinates only; get_image_data will return the untransformed data. The transformation is applied around the anchor point.

Parameters
• 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) – Degrees of clockwise rotation of the returned image. Only 90-degree increments are supported.

Return type TextureRegion

TextureRegion.save(filename=None, file=None, encoder=None)
Save this image to a file.

Parameters
• filename (str) – Used to set the image file format, and to open the output file if file is unspecified.
• file (file-like object or None) – File to write image data to.
• encoder (ImageEncoder or None) – If unspecified, all encoders matching the filename extension are tried. If all fail, the exception from the first one attempted is raised.

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

```python
class TextureSequence
```

Interface for a sequence of textures.

Typical implementations store multiple `TextureRegion`s within one `Texture` so as to minimise state changes.

**Methods:**

```python
get_animation(period[, loop])  Create an animation over this image sequence for the given constant framerate.
```

```python
get_texture_sequence()
```

**Attributes:**

```python
texture_sequence  Access this image sequence as a texture sequence.
```
Methods

TextureSequence.get_texture_sequence()

Inherited members

Methods

TextureSequence.get_animation(period, loop=True)

Create an animation over this image sequence for the given constant framerate.

:Parameters

- **period** [float] Number of seconds to display each frame.
- **loop** [bool] If True, the animation will loop continuously.

:Return
type: Animation

Note: Since pyglet 1.1

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.

Constructor:

__init__(width, height, target, id)

Methods:

- blit(x, y[, z, width, height])
- blit_into(source, x, y, z)
- blit_tiled(x, y, z, width, height) Blit this texture tiled over the given area.
- blit_to_texture(target, level, x, y[, z]) Draw this image on the currently bound texture at target.

Continued on next page
Table 2.158 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create(width, height[, internalformat, ...])</code></td>
<td>Create an empty Texture.</td>
</tr>
<tr>
<td><code>create_for_image(image)</code></td>
<td>Create a Texture with dimensions at least min_width, min_height.</td>
</tr>
<tr>
<td><code>create_for_size(target, min_width, min_height)</code></td>
<td>Delete the texture from video memory.</td>
</tr>
<tr>
<td><code>delete()</code></td>
<td>Delete the image data of this texture.</td>
</tr>
<tr>
<td><code>get_image_data([z])</code></td>
<td>Retrieve a Texture instance with all mipmap levels filled in.</td>
</tr>
<tr>
<td><code>get_region(x, y, width, height)</code></td>
<td>Create a copy of this image applying a simple transformation.</td>
</tr>
<tr>
<td><code>get_transform([flip_x, flip_y, rotate])</code></td>
<td>Save this image to a file.</td>
</tr>
</tbody>
</table>

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 Texture view of this image.
- `x`
- `y`
- `z`

Methods

**TileableTexture**.

- **blit_tiled**(x, y, z, width, height) Blit this texture tiled over the given area. The image will be tiled with the bottom-left corner of the destination rectangle aligned with the anchor point of this texture.

  **classmethod** **TileableTexture**.create_for_image(image)

- **TileableTexture**.get_region(x, y, width, height)

Inherited members

- **Methods**
  - **TileableTexture**.blit(x, y, z=0, width=None, height=None)
  - **TileableTexture**.blit_into(source, x, y, z)
  - **TileableTexture**.blit_to_texture(target, level, x, y, z=0) Draw this image on the currently bound texture at target.
This image is copied into the texture such that this image’s anchor point is aligned with the given \( x \) and \( y \) coordinates of the destination texture. If the currently bound texture is a 3D texture, the \( z \) coordinate gives the image slice to blit into.

**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*
TileableTexture.delete()
Delete the texture from video memory.

**Warning:** Deprecated. Textures are automatically released during object finalization.

TileableTexture.get_image_data(z=0)
Get the image data of this texture.
Changes to the returned instance will not be reflected in this texture.

**Parameters**
- **z** (*int*) – For 3D textures, the image slice to retrieve.

**Return type** *ImageData*

TileableTexture.get_mipmapped_texture()
Retrieve a Texture instance with all mipmap levels filled in.
Requires that image dimensions be powers of 2.

**Return type** *Texture*

**Note:** Since pyglet 1.1

TileableTexture.get_texture(rectangle=False, force_rectangle=False)
TileableTexture.get_transform(flip_x=False, flip_y=False, rotate=0)
Create a copy of this image applying a simple transformation.
The transformation is applied to the texture coordinates only; get_image_data will return the un-transformed data. The transformation is applied around the anchor point.

**Parameters**
- **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*) – Degrees of clockwise rotation of the returned image. Only 90-degree increments are supported.

**Return type** *TextureRegion*

TileableTexture.save(filename=None, file=None, encoder=None)
Save this image to a file.

**Parameters**
- **filename** (*str*) – Used to set the image file format, and to open the output file if `file` is unspecified.
- **file** (*file-like object or None*) – File to write image data to.
- **encoder** (*ImageEncoder or None*) – If unspecified, all encoders matching the filename extension are tried. If all fail, the exception from the first one attempted is raised.

**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, 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.

Methods: 

Continued on next page
get_animation(period[, loop])  Create an animation over this image sequence for the given constant framerate.
get_texture_sequence()

Attributes:

<table>
<thead>
<tr>
<th>item_height</th>
</tr>
</thead>
<tbody>
<tr>
<td>item_width</td>
</tr>
<tr>
<td>texture_sequence  Access this image sequence as a texture sequence.</td>
</tr>
</tbody>
</table>

Attributes
UniformTextureSequence.item_height
UniformTextureSequence.item_width

Inherited members

Methods

UniformTextureSequence.get_animation(period, loop=True)
Create an animation over this image sequence for the given constant framerate.

Parameters

- period  [float] Number of seconds to display each frame.
- loop  [bool] If True, the animation will loop continuously.

Return type  Animation

Note:  Since pyglet 1.1

UniformTextureSequence.get_texture_sequence()  

Attributes

UniformTextureSequence.texture_sequence
Access this image sequence as a texture sequence.

Warning:  Deprecated. Use get_texture_sequence

Type  TextureSequence

Exceptions
ImageException

Exception defined in pyglet.image

definition: ImageException

Functions

- `color_as_bytes(color)`
  - Define: `pyglet.image`

- `create(width, height[, pattern])`
  - Create an image optionally filled with the given pattern.

- `get_buffer_manager()`
  - Get the buffer manager for the current OpenGL context.

- `load(filename[, file, decoder])`
  - Load an image from a file.

- `load_animation(filename[, file, decoder])`
  - 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`  

```python
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`  

```python
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**

- `compat_platform` = `‘linux2’`
  ```python
  str(object='') -> string
  ```
  Return a nice string representation of the object. If the argument is a string, the return value is the same object.

- `division` = `_Feature((2, 2, 0, ‘alpha’, 2), (3, 0, 0, ‘alpha’, 0), 8192)`

**Notes**
Defined

- event
- gl
- glex_arb
- glu
- graphics
- key
- lib
- lib_glx
- mouse
- pprint
- 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.:

```bash
python -m pyglet.info > info.txt
```

## Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dump()</td>
<td>Dump all information to stdout.</td>
</tr>
<tr>
<td>dump_al()</td>
<td>Dump OpenAL info.</td>
</tr>
<tr>
<td>dump_avbin()</td>
<td>Dump AVbin info.</td>
</tr>
<tr>
<td>dump_gl(ctx)</td>
<td>Dump GL info.</td>
</tr>
<tr>
<td>dump_glu()</td>
<td>Dump GLU info.</td>
</tr>
<tr>
<td>dump_glx()</td>
<td>Dump GLX info.</td>
</tr>
<tr>
<td>dump_media()</td>
<td>Dump pyglet.media info.</td>
</tr>
<tr>
<td>dump_pyglet()</td>
<td>Dump pyglet version and options.</td>
</tr>
<tr>
<td>dump_python()</td>
<td>Dump Python version and environment to stdout.</td>
</tr>
<tr>
<td>dump_window()</td>
<td>Dump display, window, screen and default config info.</td>
</tr>
<tr>
<td>dump_wintab()</td>
<td>Dump WinTab info.</td>
</tr>
</tbody>
</table>

### 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.

**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
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

---

### Notes

**Defined**
- base
- 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

<table>
<thead>
<tr>
<th>drivers</th>
<th>procedural</th>
</tr>
</thead>
<tbody>
<tr>
<td>riff</td>
<td>Simple Python-only RIFF reader, supports uncompressed WAV files</td>
</tr>
</tbody>
</table>

**pyglet.media.drivers**

<table>
<thead>
<tr>
<th>silent</th>
</tr>
</thead>
</table>

**pyglet.media.drivers.silent**

<table>
<thead>
<tr>
<th>SilentAudioDriver</th>
</tr>
</thead>
</table>

Continued on next page
Table 2.171 – continued from previous page

<table>
<thead>
<tr>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SilentAudioPacket</td>
</tr>
<tr>
<td>SilentAudioPlayerPacketConsumer</td>
</tr>
<tr>
<td>SilentTimeAudioPlayer</td>
</tr>
</tbody>
</table>

Classes

```py
pyglet.media.AbstractAudioDriver -> pyglet.media.drivers.silent.SilentAudioDriver
```

**SilentAudioDriver Class**

class SilentAudioDriver

Methods:

```py
create_audio_player(source_group, player)
deflete()
```

Methods

SilentAudioDriver.create_audio_player(source_group, player)
SilentAudioDriver.delete()

Inherited members

Methods

SilentAudioDriver.get_listener()

```py
pyglet.media.drivers.silent.SilentAudioPacket
```

**SilentAudioPacket Class**

class SilentAudioPacket (timestamp, duration)

Constructor:

```py
__init__(timestamp, duration)
```

Methods:
Methods
SilentAudioPacket.consume(dt)

pyglet.media.AbstractAudioPlayer → pyglet.media.drivers.silent.SilentAudioPlayerPacketConsumer

SilentAudioPlayerPacketConsumer Class
class SilentAudioPlayerPacketConsumer (source_group, player)
  Constructor:
  __init__(source_group, player)
  Methods:
  clear()
  delete()
  get_time()
  play()
  stop()

Methods
SilentAudioPlayerPacketConsumer.clear()
SilentAudioPlayerPacketConsumer.delete()
SilentAudioPlayerPacketConsumer.get_time()
SilentAudioPlayerPacketConsumer.play()
SilentAudioPlayerPacketConsumer.stop()

Inherited members

Methods
SilentAudioPlayerPacketConsumer.set_cone_inner_angle(cone_inner_angle)
  See Player.cone_inner_angle.
SilentAudioPlayerPacketConsumer.set_cone_orientation(cone_orientation)
  See Player.cone_orientation.
SilentAudioPlayerPacketConsumer.set_cone_outer_angle(cone_outer_angle)
  See Player.cone_outer_angle.
SilentAudioPlayerPacketConsumer.\texttt{set\_cone\_outer\_gain}(cone\_outer\_gain)
See Player.cone\_outer\_gain.

SilentAudioPlayerPacketConsumer.\texttt{set\_max\_distance}(max\_distance)
See Player.max\_distance.

SilentAudioPlayerPacketConsumer.\texttt{set\_min\_distance}(min\_distance)
See Player.min\_distance.

SilentAudioPlayerPacketConsumer.\texttt{set\_pitch}(pitch)
See Player.pitch.

SilentAudioPlayerPacketConsumer.\texttt{set\_position}(position)
See Player.position.

SilentAudioPlayerPacketConsumer.\texttt{set\_volume}(volume)
See Player.volume.

\begin{itemize}
  \item \texttt{pyglet.media.AbstractAudioPlayer}
  \item \texttt{pyglet.media.drivers.silent.SilentTimeAudioPlayer}
\end{itemize}

\textbf{SilentTimeAudioPlayer Class}
\begin{verbatim}
class SilentTimeAudioPlayer (source_group, player)
    Constructor:
        \texttt{__init__}(source_group, player)
        Create a new audio player.

        Parameters
        \begin{itemize}
            \item source\_group (SourceGroup) – Source group to play from.
            \item player (Player) – Player to receive EOS and video frame sync events.
        \end{itemize}

        Methods:
        \begin{itemize}
            \item \texttt{clear}()
            \item \texttt{delete}()
            \item \texttt{get\_time}()
            \item \texttt{play}()
            \item \texttt{stop}()
        \end{itemize}
\end{verbatim}
Inherited members

Methods

SilentTimeAudioPlayer.set_cone_inner_angle(cone_inner_angle)
    See Player.cone_inner_angle.

SilentTimeAudioPlayer.set_cone_orientation(cone_orientation)
    See Player.cone_orientation.

SilentTimeAudioPlayer.set_cone_outer_angle(cone_outer_angle)
    See Player.cone_outer_angle.

SilentTimeAudioPlayer.set_cone_outer_gain(cone_outer_gain)
    See Player.cone_outer_gain.

SilentTimeAudioPlayer.set_max_distance(max_distance)
    See Player.max_distance.

SilentTimeAudioPlayer.set_min_distance(min_distance)
    See Player.min_distance.

SilentTimeAudioPlayer.set_pitch(pitch)
    See Player.pitch.

SilentTimeAudioPlayer.set_position(position)
    See Player.position.

SilentTimeAudioPlayer.set_volume(volume)
    See Player.volume.

create_audio_driver()

Functions

create_audio_driver Function  Defined in pyglet.media.drivers.silent
create_audio_driver()
Classes

Table 2.177 – continued from previous page

<table>
<thead>
<tr>
<th>Square</th>
</tr>
</thead>
<tbody>
<tr>
<td>WhiteNoise</td>
</tr>
</tbody>
</table>

ProceduralSource Class

class ProceduralSource (duration, sample_rate=44800, sample_size=16)

Constructor:

__init__(duration, sample_rate=44800, sample_size=16)

Methods:

- get_audio_data(bytes)
- seek(timestamp)

Attributes:

- audio_format
- duration The length of the source, in seconds.
- info
- video_format

Methods

ProceduralSource.get_audio_data(bytes)
ProceduralSource.seek(timestamp)

Inherited members

Methods

ProceduralSource.get_animation()

Import all video frames into memory as an Animation.

An empty animation will be returned if the source has no video. Otherwise, the animation will contain all unplayed video frames (the entire source, if it has not been queued on a player). After creating the animation, the source will be at EOS.
This method is unsuitable for videos running longer than a few seconds.

**Note:** Since pyglet 1.1

**Return type** `pyglet.image.Animation`

`ProceduralSource.get_next_video_frame()`

Get the next video frame.

Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

**Note:** Since pyglet 1.1

**Return type** `pyglet.image.AbstractImage`

**Returns** The next video frame image, or `None` if the video frame could not be decoded or there are no more video frames.

`ProceduralSource.get_next_video_timestamp()`

Get the timestamp of the next video frame.

**Note:** Since pyglet 1.1

**Return type** `float`

**Returns** The next timestamp, or `None` if there are no more video frames.

`ProceduralSource.play()`

Play the source.

This is a convenience method which creates a `Player` for this source and plays it immediately.

**Return type** `Player`

**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`
**Saw Class**

class Saw (duration, frequency=440, **kwargs)

**Constructor:**

```python
__init__ (duration, frequency=440, **kwargs)
```

**Methods:**

- `get_audio_data(bytes)`
- `seek(timestamp)`

**Attributes:**

- `audio_format`
- `duration` The length of the source, in seconds.
- `info`
- `video_format`

**Inherited members**

**Methods**

*Saw.get_animation()*

Import all video frames into memory as an Animation.

An empty animation will be returned if the source has no video. Otherwise, the animation will contain all unplayed video frames (the entire source, if it has not been queued on a player). After creating the animation, the source will be at EOS.

This method is unsuitable for videos running longer than a few seconds.

**Note:** Since pyglet 1.1

**Return type** `pyglet.image.Animation`

*Saw.get_audio_data(bytes)*

*Saw.get_next_video_frame()*

Get the next video frame.

Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

**Note:** Since pyglet 1.1

**Return type** `pyglet.image.AbstractImage`

**Returns** The next video frame image, or `None` if the video frame could not be decoded or there are no more video frames.
get_next_video_timestamp()  
Get the timestamp of the next video frame.

Note: Since pyglet 1.1

Return type float

Returns The next timestamp, or None if there are no more video frames.

play()  
Play the source.

This is a convenience method which creates a Player for this source and plays it immediately.

Return type Player

seek(timestamp)

Attributes

audio_format = None

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

info = None

video_format = None

Silence Class

class Silence(duration, sample_rate=44800, sample_size=16)

Constructor:

__init__(duration, sample_rate=44800, sample_size=16)

Methods:

get_audio_data(bytes)

seek(timestamp)

Attributes:

audio_format

Continued on next page
Table 2.183 – continued from previous page

<table>
<thead>
<tr>
<th>duration</th>
<th>The length of the source, in seconds.</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td></td>
</tr>
<tr>
<td>video_format</td>
<td></td>
</tr>
</tbody>
</table>

Inherited members

Methods

Silence.get_animation()
Import all video frames into memory as an Animation.

An empty animation will be returned if the source has no video. Otherwise, the animation will contain all unplayed video frames (the entire source, if it has not been queued on a player). After creating the animation, the source will be at EOS.

This method is unsuitable for videos running longer than a few seconds.

Note: Since pyglet 1.1

Return type pyglet.image.Animation

Silence.get_audio_data(bytes)

Silence.get_next_video_frame()
Get the next video frame.

Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

Note: Since pyglet 1.1

Return type pyglet.image.AbstractImage

Returns The next video frame image, or None if the video frame could not be decoded or there are no more video frames.

Silence.get_next_video_timestamp()
Get the timestamp of the next video frame.

Note: Since pyglet 1.1

Return type float

Returns The next timestamp, or None if there are no more video frames.

Silence.play()
Play the source.

This is a convenience method which creates a Player for this source and plays it immediately.

Return type Player

Silence.seek(timestamp)
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.info = None
Silence.video_format = None

Sine Class
class Sine(duration, frequency=440, **kwargs)

Constructor:
  __init__ (duration, frequency=440, **kwargs)

Methods:

  __init__
  get_audio_data(bytes)
  seek(timestamp)

Attributes:

  __init__
  audio_format
  duration  The length of the source, in seconds.
  info
  video_format

Inherited members

Methods

Sine.get_animation()
  Import all video frames into memory as an Animation.
  An empty animation will be returned if the source has no video. Otherwise, the animation will contain all unplayed video frames (the entire source, if it has not been queued on a player). After creating the animation, the source will be at EOS.
This method is unsuitable for videos running longer than a few seconds.

Note: Since pyglet 1.1

Return type  pyglet.image.Animation

Sine.get_audio_data(bytes)

Sine.get_next_video_frame()

Get the next video frame.

Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

Note: Since pyglet 1.1

Return type  pyglet.image.AbstractImage

Returns The next video frame image, or None if the video frame could not be decoded or there are no more video frames.

Sine.get_next_video_timestamp()

Get the timestamp of the next video frame.

Note: Since pyglet 1.1

Return type  float

Returns The next timestamp, or None if there are no more video frames.

Sine.play()

Play the source.

This is a convenience method which creates a Player for this source and plays it immediately.

Return type  Player

Sine.seek(timestamp)

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.info = None

Sine.video_format = None
Square Class

class Square (duration, frequency=440, **kwargs)

Constructor:

__init__ (duration, frequency=440, **kwargs)

Methods:

- get_audio_data(bytes)
- seek(timestamp)

Attributes:

- audio_format
- duration  The length of the source, in seconds.
- info
- video_format

Inherited members

Methods

Square.get_animation ()

Import all video frames into memory as an Animation.

An empty animation will be returned if the source has no video. Otherwise, the animation will contain all unplayed video frames (the entire source, if it has not been queued on a player). After creating the animation, the source will be at EOS.

This method is unsuitable for videos running longer than a few seconds.

Note: Since pyglet 1.1

Return type  pyglet.image.Animation

Square.get_audio_data (bytes)

Square.get_next_video_frame ()

Get the next video frame.

Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

Note: Since pyglet 1.1
Return type  `pyglet.image.AbstractImage`

Returns  The next video frame image, or `None` if the video frame could not be decoded or there are no more video frames.

```py
Square.get_next_video_timestamp()
```
Get the timestamp of the next video frame.

Note: Since pyglet 1.1

Return type  `float`

Returns  The next timestamp, or `None` if there are no more video frames.

```py
Square.play()
```
Play the source.

This is a convenience method which creates a Player for this source and plays it immediately.

Return type  `Player`

```py
Square.seek(timestamp)
```

Attributes

```py
Square.audio_format = None
```

```py
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`

```py
Square.info = None
```

```py
Square.video_format = None
```

---

**WhiteNoise Class**

class WhiteNoise  

```py
class WhiteNoise(duration, sample_rate=44800, sample_size=16)
```

Constructor:

```py
__init__(duration, sample_rate=44800, sample_size=16)
```

Methods:

```py
get_audio_data(bytes)
```
```py
seek(timestamp)
```

Attributes:
audio_format
duration The length of the source, in seconds.
info
video_format

Inherited members

Methods

WhiteNoise.get_animation()

Import all video frames into memory as an Animation.

An empty animation will be returned if the source has no video. Otherwise, the animation will contain all unplayed video frames (the entire source, if it has not been queued on a player). After creating the animation, the source will be at EOS.

This method is unsuitable for videos running longer than a few seconds.

Note: Since pyglet 1.1

Return type  pyglet.image.Animation

WhiteNoise.get_audio_data(bytes)

WhiteNoise.get_next_video_frame()

Get the next video frame.

Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

Note: Since pyglet 1.1

Return type  pyglet.image(AbstractImage

Returns  The next video frame image, or None if the video frame could not be decoded or there are no more video frames.

WhiteNoise.get_next_video_timestamp()

Get the timestamp of the next video frame.

Note: Since pyglet 1.1

Return type  float

Returns  The next timestamp, or None if there are no more video frames.

WhiteNoise.play()

Play the source.

This is a convenience method which creates a Player for this source and plays it immediately.

Return type  Player

WhiteNoise.
WhiteNoise.seek(timestamp)

**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.info = None

WhiteNoise.video_format = None

---

**Defined**

**Notes**

- math
- os

---

**pyglet.media.riff**  Simple Python-only RIFF reader, supports uncompressed WAV files.

---

**Classes**

---

**RIFFChunk Class**

class RIFFChunk (file, name, length, offset)

  Constructor:

  __init__(file, name, length, offset)

  Methods:
get_data()

Attributes:

header_fmt
header_length

Methods

RIFFChunk.get_data()

Attributes

RIFFChunk.header_fmt = ‘<4sI’
RIFFChunk.header_length = 8

RIFFFile Class

class RIFFFile(file)

Constructor:

__init__(file)

Methods:

get_chunks()
get_wave_form()

Methods

RIFFFile.get_wave_form()

Inherited members

Methods

RIFFFile.get_chunks()
RIFFForm Class

class RIFFForm(file, offset)

   Constructor:
___init___(file, offset)

   Methods:
       get_chunks()

Methods

RIFFForm.get_chunks()

RIFFType Class

class RIFFType(*args, **kwargs)

   Constructor:
___init___(*args, **kwargs)

   Methods:
       get_data()

   Attributes:
       header_fmt
       header_length

Inherited members
Methods

RIFFType.get_data()

Attributes

RIFFType.header_fmt = '<4sL'
RIFFType.header_length = 8

WaveDataChunk Class

class WaveDataChunk(file, name, length, offset)

Constructor:

__init__(file, name, length, offset)

Methods:

get_data()

Attributes:

header_fmt
header_length

Inherited members

Methods

WaveDataChunk.get_data()

Attributes

WaveDataChunk.header_fmt = '<4sL'
WaveDataChunk.header_length = 8
**WaveForm Class**

class WaveForm(file, offset)  
Constructor:  
    __init__(file, offset)  
Methods:  
    get_chunks()  
    get_data_chunk()  
    get_format_chunk()  

**Methods**
WaveForm.get_data_chunk()  
WaveForm.get_format_chunk()  

**Inherited members**

**Methods**
WaveForm.get_chunks()  

**WaveFormatChunk Class**

class WaveFormatChunk(*args, **kwargs)  
Constructor:  
    __init__(*args, **kwargs)  
Methods:  
    get_data()  

2.1. pyglet
Attributes:

- header_fmt
- header_length

Inherited members

Methods

WaveFormatChunk.get_data()

Attributes

WaveFormatChunk.header_fmt = '<4sL'
WaveFormatChunk.header_length = 8

WaveSource Class

class WaveSource (filename, file=None)

Constructor:

__init__ (filename, file=None)

Methods:

- get_audio_data(bytes)
- seek(timestamp)

Attributes:

- audio_format
- duration
- info
- is_queued
- video_format

The length of the source, in seconds.

Determine if this source has been queued on a Player yet.
Methods

`WaveSource.get_audio_data (bytes)`

`WaveSource.seek (timestamp)`

Inherited members

Methods

`WaveSource.get_animation()`  
Import all video frames into memory as an Animation.

An empty animation will be returned if the source has no video. Otherwise, the animation will contain all unplayed video frames (the entire source, if it has not been queued on a player). After creating the animation, the source will be at EOS.

This method is unsuitable for videos running longer than a few seconds.

**Note:** Since pyglet 1.1

**Return type** `pyglet.image.Animation`

`WaveSource.get_next_video_frame()`  
Get the next video frame.

Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

**Note:** Since pyglet 1.1

**Return type** `pyglet.image.AbstractImage`

**Returns** The next video frame image, or None if the video frame could not be decoded or there are no more video frames.

`WaveSource.get_next_video_timestamp()`  
Get the timestamp of the next video frame.

**Note:** Since pyglet 1.1

**Return type** `float`

**Returns** The next timestamp, or None if there are no more video frames.

`WaveSource.play()`  
Play the source.

This is a convenience method which creates a Player for this source and plays it immediately.

**Return type** `Player`

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

Exception defined in pyglet.media.riff

WAVEFormatException

Exception defined in pyglet.media.riff

Variables

IBM_FORMAT_ADPCM = 259

int(x=0) -> int or long int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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) -> int or long int(x, base=10) -> int or long
Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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**

```python
int(x=0) -> int or long int(x, base=10) -> int or long
```

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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**

```python
int(x=0) -> int or long int(x, base=10) -> int or long
```

Convert a number or string to an integer, or return 0 if no arguments are given. If x is floating point, the conversion truncates towards zero. If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or Unicode object 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**

  - struct

### Classes

- **AVbinSourceLoader**
- **AbstractAudioDriver**
- **AbstractAudioPlayer** Base class for driver audio players.
- **AbstractListener** The listener properties for positional audio.
- **AbstractSourceLoader**
- **AudioData** A single packet of audio data.
- **AudioFormat** Audio details.
- **ManagedSoundPlayer** Warning: Deprecated. Use **Player**
- **MediaEvent**
- **MediaThread** A thread that cleanly exits on interpreter shutdown, and provides a sleep method that can be interrupted.
- **Player** High-level sound and video player.
- **PlayerGroup** Group of players that can be played and paused simultaneously.
- **RIFFSourceLoader**
- **Source** An audio and/or video source.
- **SourceGroup** Read data from a queue of sources, with support for looping.
Table 2.205 – continued from previous page

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceInfo</td>
<td>Source metadata information.</td>
</tr>
<tr>
<td>StaticMemorySource</td>
<td>Helper class for default implementation of StaticSource.</td>
</tr>
<tr>
<td>StaticSource</td>
<td>A source that has been completely decoded in memory.</td>
</tr>
<tr>
<td>StreamingSource</td>
<td>A source that is decoded as it is being played, and can only be queued once.</td>
</tr>
<tr>
<td>VideoFormat</td>
<td>Video details.</td>
</tr>
<tr>
<td>WorkerThread</td>
<td></td>
</tr>
</tbody>
</table>

**AVbinSourceLoader Class**

class AVbinSourceLoader

Methods:

```python
load(filename, file)
```

**AbstractAudioDriver Class**

class AbstractAudioDriver

Methods:

```python
create_audio_player(source_group, player)
get_listener()
```

Methods

```python
AbstractAudioDriver.create_audio_player(source_group, player)
AbstractAudioDriver.get_listener()
```
AbstractAudioPlayer Class

class AbstractAudioPlayer(source_group, player)
    Base class for driver audio players.

    Constructor:
    __init__(source_group, player)
        Create a new audio player.

        Parameters
        • source_group (SourceGroup) – Source group to play from.
        • player (Player) – Player to receive EOS and video frame sync events.

    Methods:
    clear() Clear all buffered data and prepare for replacement data.
    delete() Stop playing and clean up all resources used by player.
    get_time() Return approximation of current playback time within current source.
    play() Begin playback.
    set_max_distance(max_distance) See Player.max_distance.
    set_min_distance(min_distance) See Player.min_distance.
    set_pitch(pitch) See Player.pitch.
    set_position(position) See Player.position.
    set_volume(volume) See Player.volume.
    stop() Stop (pause) playback.

Methods
AbstractAudioPlayer.clear() Clear all buffered data and prepare for replacement data.
The player should be stopped before calling this method.

AbstractAudioPlayer.delete() Stop playing and clean up all resources used by player.

AbstractAudioPlayer.get_time() Return approximation of current playback time within current source.
Returns None if the audio player does not know what the playback time is (for example, before any valid audio data has been read).

Return type float
Returns current play cursor time, in seconds.

AbstractAudioPlayer\texttt{.play}()

Begin playback.

AbstractAudioPlayer\texttt{.set\_cone\_inner\_angle}(cone\_inner\_angle)

See Player.cone_inner_angle.

AbstractAudioPlayer\texttt{.set\_cone\_orientation}(cone\_orientation)

See Player.cone_orientation.

AbstractAudioPlayer\texttt{.set\_cone\_outer\_angle}(cone\_outer\_angle)

See Player.cone_outer_angle.

AbstractAudioPlayer\texttt{.set\_cone\_outer\_gain}(cone\_outer\_gain)

See Player.cone_outer_gain.

AbstractAudioPlayer\texttt{.set\_max\_distance}(max\_distance)

See Player.max_distance.

AbstractAudioPlayer\texttt{.set\_min\_distance}(min\_distance)

See Player.min_distance.

AbstractAudioPlayer\texttt{.set\_pitch}(pitch)

See Player.pitch.

AbstractAudioPlayer\texttt{.set\_position}(position)

See Player.position.

AbstractAudioPlayer\texttt{.set\_volume}(volume)

See Player.volume.

AbstractAudioPlayer\texttt{.stop}()

Stop (pause) playback.

---

\texttt{pyglet.media.AbstractListener}

\textbf{AbstractListener Class}

class \texttt{AbstractListener}

The listener properties for positional audio.

You can obtain the singleton instance of this class by calling AbstractAudioDriver.get_listener.

Attributes:

\begin{tabular}{|l|l|p{0.7\textwidth}|}
\hline
\texttt{forward\_orientation} & A vector giving the direction the listener is facing. \\
\texttt{position} & The position of the listener in 3D space. \\
\texttt{up\_orientation} & A vector giving the “up” orientation of the listener. \\
\texttt{volume} & The master volume for sound playback. \\
\hline
\end{tabular}

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

AbstractSourceLoader Class

class AbstractSourceLoader

Methods:

load(filename, file)

Methods

AbstractSourceLoader.load(filename, file)
**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.

**Constructor:**

```python
def __init__(self, data, length, timestamp, duration, events)
```

**Methods:**

- `consume(bytes, audio_format)` Remove some data from beginning of packet.
- `get_string_data()` Return data as a string.

---

**Methods**

- `AudioData.consume (bytes, audio_format)` Remove some data from beginning of packet. All events are cleared.
- `AudioData.get_string_data ()` Return data as a string. (Python 3: return as bytes)

---

**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).

Constructor:

```python
__init__(channels, sample_size, sample_rate)
```

---

**ManagedSoundPlayer** Class

class **ManagedSoundPlayer** (*args, **kwargs)

Warning: Deprecated. Use **Player**

Constructor:

```python
__init__(*args, **kwargs)
```

Methods:

```python
delete()
ge_get_texture()
next()
next_source()
pause()
play()
queue(source)
seek(time)
seek_next_frame()  # Step forwards one video frame in the current Source.
update_texture([dt, time])
```

Events:

```python
on_eos()
on_player_eos()  # The player ran out of sources.
on_source_group_eos()  # The current source group ran out of data.
```

Attributes:

- **EOS_LOOP**
- **EOS_NEXT**

Continued on next page
Table 2.214 – continued from previous page

| EOS_PAUSE            |
| EOS_STOP             |
| cone_inner_angle     |
| cone_orientation     |
| cone_outer_angle     |
| cone_outer_gain      |
| eos_action           |
| event_types          |
| max_distance         |
| min_distance         |
| pitch                |
| playing              |
| position             |
| source               |
| time                 |
| volume               |

Inherited members

**Methods**

ManagedSoundPlayer.delete()  
ManagedSoundPlayer.dispatch_event(event_type, *args)  
  Dispatch a single event to the attached handlers.  
  
  The event is propagated to all handlers from from the top of the stack until one returns EVENT_HANDLED. This method should be used only by EventDispatcher implementors; applications should call the dispatch_events method.  
  
  Since pyglet 1.2, the method returns EVENT_HANDLED if an event handler returned EVENT_HANDLED or EVENT_UNHANDLED if all events returned EVENT_UNHANDLED. If no matching event handlers are in the stack, False is returned.  

  **Parameters**  
  
  - event_type (str) – Name of the event.  
  - args (sequence) – Arguments to pass to the event handler.  

  **Return type** bool or None  

  **Returns** (Since pyglet 1.2) EVENT_HANDLED if an event handler returned EVENT_HANDLED; EVENT_UNHANDLED if one or more event handlers were invoked but returned only EVENT_UNHANDLED; otherwise False. In pyglet 1.1 and earlier, the return value is always None.

ManagedSoundPlayer.event(*args)  
  Function decorator for an event handler.  

  **Usage:**
```python
win = window.Window()

@win.event
def on_resize(self, width, height):
    # ...

or:

@win.event('on_resize')
def foo(self, width, height):
    # ...
```

ManagedSoundPlayer.get_texture()
ManagedSoundPlayer.next()
ManagedSoundPlayer.next_source()
ManagedSoundPlayer.pause()
ManagedSoundPlayer.play()
ManagedSoundPlayer.pop_handlers()
Pop the top level of event handlers off the stack.
ManagedSoundPlayer.push_handlers(*args, **kwargs)
Push a level onto the top of the handler stack, then attach zero or more event handlers.
If keyword arguments are given, they name the event type to attach. Otherwise, a callable’s __name__ attribute will be used. Any other object may also be specified, in which case it will be searched for callables with event names.
ManagedSoundPlayer.queue(source)
ManagedSoundPlayer.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.

ManagedSoundPlayer.remove_handler(name, handler)
Remove a single event handler.
The given event handler is removed from the first handler stack frame it appears in. The handler must be the exact same callable as passed to set_handler, set_handlers or push_handlers; and the name must match the event type it is bound to.
No error is raised if the event handler is not set.

Parameters

- **name** (str) – Name of the event type to remove.
- **handler** (callable) – Event handler to remove.

ManagedSoundPlayer.remove_handlers(*args, **kwargs)
Remove event handlers from the event stack.
See push_handlers for the accepted argument types. All handlers are removed from the first stack frame that contains any of the given handlers. No error is raised if any handler does not appear in that frame, or if no stack frame contains any of the given handlers.
If the stack frame is empty after removing the handlers, it is removed from the stack. Note that this interferes with the expected symmetry of `push_handlers` and `pop_handlers`.

```py
ManagedSoundPlayer.seek(time)
```

Step forwards one video frame in the current Source.

```py
ManagedSoundPlayer.seek_next_frame()
```

Attach a single event handler.

```py
Parameters

- name (str) – Name of the event type to attach to.
- handler (callable) – Event handler to attach.
```

```py
ManagedSoundPlayer.set_handlers(*args, **kwargs)
```

Attach one or more event handlers to the top level of the handler stack.

See `push_handlers` for the accepted argument types.

```py
ManagedSoundPlayer.update_texture(dt=None, time=None)
```

**Events**

```py
ManagedSoundPlayer.on_eos()
```

```py
ManagedSoundPlayer.on_player_eos()
```

The player ran out of sources.

```py
ManagedSoundPlayer.on_source_group_eos()
```

The current source group ran out of data.

The default behaviour is to advance to the next source group if possible.

**Attributes**

```py
ManagedSoundPlayer.EOS_LOOP = 'loop'
```

```py
ManagedSoundPlayer.EOS_NEXT = 'next'
```

```py
ManagedSoundPlayer.EOS_PAUSE = 'pause'
```

```py
ManagedSoundPlayer.EOS_STOP = 'stop'
```

```py
ManagedSoundPlayer.cone_inner_angle
```

```py
ManagedSoundPlayer.cone_orientation
```

```py
ManagedSoundPlayer.cone_outer_angle
```

```py
ManagedSoundPlayer.cone_outer_gain
```

```py
ManagedSoundPlayer.eos_action
```

Set the behaviour of the player when it reaches the end of the current source.

This must be one of the constants `EOS_NEXT`, `EOS_PAUSE`, `EOS_STOP` or `EOS_LOOP`.

**Warning:** Deprecated. Use `SourceGroup.loop` and `SourceGroup.advance_after_eos`

```py
Type str
```
ManagedSoundPlayer.event_types = ['on_eos', 'on_player_eos', 'on_source_group_eos']
ManagedSoundPlayer.max_distance
ManagedSoundPlayer.min_distance
ManagedSoundPlayer.pitch
ManagedSoundPlayer.playing
ManagedSoundPlayer.position
ManagedSoundPlayer.source
ManagedSoundPlayer.time
ManagedSoundPlayer.volume

MediaEvent Class

class MediaEvent (timestamp, event, *args)
    Constructor:
    __init__ (timestamp, event, *args)

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.
    Constructor:
    __init__ (target=None)
    Methods:
    notify() Interrupt the current sleep operation.
    run()
    Continued on next page
Methods
MediaThread.notify()
Interrupt the current sleep operation.
   If the thread is currently sleeping, it will be woken immediately, instead of waiting the full duration of the timeout.
MediaThread.run()
MediaThread.sleep(timeout)
   Wait for some amount of time, or until notified.

   Parameters timeout (float) – Time to wait, in seconds.

MediaThread.start()
MediaThread.stop()
   Stop the thread and wait for it to terminate.
   The stop instance variable is set to True and the condition is notified. It is the responsibility of the run method to check the value of stop after each sleep or wait and to return if set.

Player Class
class Player
   High-level sound and video player.

   Constructor:
   __init__()

   Methods:
delete()
get_texture()
next()
   Warning: Deprecated. Use next_source instead.
next_source()
pause()
play()
queue(source)
seek(time)
Table 2.216 – continued from previous page

seek_next_frame() Step forwards one video frame in the current Source.
update_texture([dt, time])

Events:

- on_eos()
- on_player_eos() The player ran out of sources.
- on_source_group_eos() The current source group ran out of data.

Attributes:

- EOS_LOOP The player will loop the current stream continuously.
- EOS_NEXT The player will move on to the next queued stream when it reaches the end of the current source.
- EOS_PAUSE The player will pause when it reaches the end of the stream.
- EOS_STOP The player will stop entirely; valid only for ManagedSoundPlayer.
- cone_inner_angle
- cone_orientation
- cone_outer_angle
- cone_outer_gain
- eos_action Set the behaviour of the player when it reaches the end of the current source.
- event_types
- max_distance
- min_distance
- pitch
- playing
- position
- source
- time
- volume

Methods

Player.delete()
Player.get_texture()
Player.next()

Warning: Deprecated. Use next_source instead.

Player.next_source()
Player.pause()
Player.play()
Player.queue(source)
Player.seek(time)
Player.\texttt{seek\_next\_frame}()
Step forwards one video frame in the current Source.

Player.\texttt{update\_texture}( \texttt{dt=None, time=None} )

\textbf{Events}
Player.\texttt{on\_eos}()
Player.\texttt{on\_player\_eos}()
The player ran out of sources.

Player.\texttt{on\_source\_group\_eos}()
The current source group ran out of data.
The default behaviour is to advance to the next source group if possible.

\textbf{Attributes}
Player.\texttt{EOS\_LOOP} = \texttt{`loop`}
The player will loop the current stream continuously.

\begin{verbatim}
Warning: Deprecated. Use SourceGroup.loop
\end{verbatim}

Player.\texttt{EOS\_NEXT} = \texttt{`next`}
The player will move on to the next queued stream when it reaches the end of the current source. If there is no source queued, the player will pause.

\begin{verbatim}
Warning: Deprecated. Use SourceGroup.advance\_after\_eos
\end{verbatim}

Player.\texttt{EOS\_PAUSE} = \texttt{`pause`}
The player will pause when it reaches the end of the stream.

\begin{verbatim}
Warning: Deprecated. Use SourceGroup.advance\_after\_eos
\end{verbatim}

Player.\texttt{EOS\_STOP} = \texttt{`stop`}
The player will stop entirely; valid only for ManagedSoundPlayer.

\begin{verbatim}
Warning: Deprecated. Use SourceGroup.advance\_after\_eos
\end{verbatim}

Player.\texttt{cone\_inner\_angle}
Player.\texttt{cone\_orientation}
Player.\texttt{cone\_outer\_angle}
Player.\texttt{cone\_outer\_gain}
Player.\texttt{eos\_action}
Set the behaviour of the player when it reaches the end of the current source.
This must be one of the constants \texttt{EOS\_NEXT, EOS\_PAUSE, EOS\_STOP} or \texttt{EOS\_LOOP}.

\begin{verbatim}
Warning: Deprecated. Use SourceGroup.loop and SourceGroup.advance\_after\_eos
\end{verbatim}

\texttt{Type str}

Player.\texttt{event\_types} = [\texttt{`on\_eos`, `on\_player\_eos`, `on\_source\_group\_eos`}]

Player.\texttt{max\_distance}
Player.min_distance
Player.pitch
Player.playing
Player.position
Player.source
Player.time
Player.volume

Inherited members

Methods

Player.dispatch_event(event_type, *args)
Dispatch a single event to the attached handlers.

The event is propagated to all handlers from from the top of the stack until one returns EVENT_HANDLED. This method should be used only by EventDispatcher implementors; applications should call the dispatch_events method.

Since pyglet 1.2, the method returns EVENT_HANDLED if an event handler returned EVENT_HANDLED or EVENT_UNHANDLED if one or more event handlers were invoked but returned only EVENT_UNHANDLED; otherwise False. In pyglet 1.1 and earlier, the return value is always None.

Parameters

• event_type (str) – Name of the event.
• args (sequence) – Arguments to pass to the event handler.

Return type
bool or None

Returns
(Since pyglet 1.2) EVENT_HANDLED if an event handler returned EVENT_HANDLED; EVENT_UNHANDLED if one or more event handlers were invoked but returned only EVENT_UNHANDLED; otherwise False. In pyglet 1.1 and earlier, the return value is always None.

Player.event(*args)
Function decorator for an event handler.

Usage:

win = window.Window()

@win.event
def on_resize(self, width, height):
    # ...

or:

@win.event('on_resize')
def foo(self, width, height):
    # ...

Player.pop_handlers()
Pop the top level of event handlers off the stack.
Player.push_handlers(*args, **kwargs)
Push a level onto the top of the handler stack, then attach zero or more event handlers.

If keyword arguments are given, they name the event type to attach. Otherwise, a callable’s
__name__ attribute will be used. Any other object may also be specified, in which case it will
be searched for callables with event names.

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.

Player.remove_handler(name, handler)
Remove a single event handler.

The given event handler is removed from the first handler stack frame it appears in. The handler
must be the exact same callable as passed to set_handler, set_handlers or push_handlers; and the
name must match the event type it is bound to.

No error is raised if the event handler is not set.

Parameters

• name (str) – Name of the event type to remove.
• handler (callable) – Event handler to remove.

Player.remove_handlers(*args, **kwargs)
Remove event handlers from the event stack.

See push_handlers for the accepted argument types. All handlers are removed from the first stack
frame that contains any of the given handlers. No error is raised if any handler does not appear in
that frame, or if no stack frame contains any of the given handlers.

If the stack frame is empty after removing the handlers, it is removed from the stack. Note that this
interferes with the expected symmetry of push_handlers and pop_handlers.

Player.set_handler(name, handler)
Attach a single event handler.

Parameters

• name (str) – Name of the event type to attach to.
• handler (callable) – Event handler to attach.

Player.set_handlers(*args, **kwargs)
Attach one or more event handlers to the top level of the handler stack.

See push_handlers for the accepted argument types.
class **PlayerGroup** *(players)*

Group of players that can be played and paused simultaneously.

**Variables** `players` – Players in this group.

**Constructor:**

```python
__init__(players)
```

Create a player group for the given set of players.

All players in the group must currently not belong to any other group.

**Parameters** `players` *(Sequence of Player)* – Players to add to this group.

**Methods:**

```python
pause()
```

Pause all players in the group simultaneously.

```python
play()
```

Begin playing all players in the group simultaneously.

---

**Methods**

`PlayerGroup.pause()`  
Pause all players in the group simultaneously.

`PlayerGroup.play()`  
Begin playing all players in the group simultaneously.

---

**RIFFSourceLoader Class**

class **RIFFSourceLoader**

**Methods:**

```python
load(filename, file)
```

**Methods**

`RIFFSourceLoader.load(filename, file)`
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

Methods:

- `get_animation()` Import all video frames into memory as an Animation.
- `get_audio_data(bytes)` Get next packet of audio data.
- `get_next_video_frame()` Get the next video frame.
- `get_next_video_timestamp()` Get the timestamp of the next video frame.
- `play()` Play the source.
- `seek(timestamp)` Seek to given timestamp.

Attributes:

- `audio_format`
- `duration` The length of the source, in seconds.
- `info`
- `video_format`

Methods

Source.

**get_animation()**

Import all video frames into memory as an Animation.

An empty animation will be returned if the source has no video. Otherwise, the animation will contain all unplayed video frames (the entire source, if it has not been queued on a player). After creating the animation, the source will be at EOS.

This method is unsuitable for videos running longer than a few seconds.

Note: Since pyglet 1.1

Return type **pyglet.image.Animation**
Source. `get_audio_data(bytes)`
Get next packet of audio data.

**Parameters**
- `bytes (int)` – Maximum number of bytes of data to return.

**Return type** `AudioData`

**Returns**
Next packet of audio data, or None if there is no (more) data.

Source. `get_next_video_frame()`
Get the next video frame.

Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

**Note:** Since pyglet 1.1

**Return type** `pyglet.image.AbstractImage`

**Returns**
The next video frame image, or None if the video frame could not be decoded or there are no more video frames.

Source. `get_next_video_timestamp()`
Get the timestamp of the next video frame.

**Note:** Since pyglet 1.1

**Return type** `float`

**Returns**
The next timestamp, or None if there are no more video frames.

Source. `play()`
Play the source.

This is a convenience method which creates a Player for this source and plays it immediately.

**Return type** `Player`

Source. `seek(timestamp)`
Seek to given timestamp.

**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`
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.

Constructor:

__init__ (audio_format, video_format)

Methods:

get_audio_data (bytes) Get next audio packet.

get_current_source ()

get_next_video_frame () Get the next video frame.

get_next_video_timestamp () Get the timestamp of the next video frame.

has_next ()

next ([immediate])

next_source ([immediate])

queue (source)

seek (time)

translate_timestamp (timestamp) Get source-relative timestamp for the audio player’s timestamp.

Attributes:

loop Loop the current source indefinitely or until next is called.

Methods

SourceGroup . get_audio_data (bytes)

Get next audio packet.

Parameters bytes (int) – Hint for preferred size of audio packet; may be ignored.

Return type AudioData

Returns Audio data, or None if there is no more data.

SourceGroup . get_current_source ()

SourceGroup . get_next_video_frame ()

Get the next video frame.

Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

Return type pyglet.image.AbstractImage
**Returns**  The next video frame image, or None if the video frame could not be decoded or there are no more video frames.

SourceGroup.get_next_video_timestamp()
Get the timestamp of the next video frame.

**Return type**  float

**Returns**  The next timestamp, or None if there are no more video frames.

SourceGroup.has_next()

SourceGroup.next(immediate=True)

**Warning:** Deprecated. Use next_source instead.

SourceGroup.next_source(immediate=True)

SourceGroup.queue(source)

SourceGroup.seek(time)

SourceGroup.translate_timestamp(timestamp)
Get source-relative timestamp for the audio player’s timestamp.

**Attributes**

SourceGroup.loop
Loop the current source indefinitely or until next is called. Initially False.

**Type**  bool

### 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:

```
album
author
comment
copyright
genre
title
track
year
```

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.

Constructor:
```
__init__(data, audio_format)
```

Construct a memory source over the given data buffer.

Methods:
```
get_animation() Import all video frames into memory as an Animation.
get_audio_data(bytes) Get the next video frame.
get_next_video_timestamp() Get the timestamp of the next video frame.
play() Play the source.
seek(timestamp)
```
Attributes:

- `audio_format`
- `duration` The length of the source, in seconds.
- `info`
- `video_format`

Methods

*StaticMemorySource*.

- `get_audio_data(bytes)`
- `seek(timestamp)`

Inherited members

Methods

*StaticMemorySource*.

- `get_animation()`
  
  Import all video frames into memory as an *Animation*.
  
  An empty animation will be returned if the source has no video. Otherwise, the animation will contain all unplayed video frames (the entire source, if it has not been queued on a player). After creating the animation, the source will be at EOS.
  
  This method is unsuitable for videos running longer than a few seconds.

  **Note:** Since pyglet 1.1

  **Return type** *pyglet.image.Animation*

- `get_next_video_frame()`
  
  Get the next video frame.
  
  Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

  **Note:** Since pyglet 1.1

  **Return type** *pyglet.image.AbstractImage*

  **Returns** The next video frame image, or *None* if the video frame could not be decoded or there are no more video frames.

- `get_next_video_timestamp()`
  
  Get the timestamp of the next video frame.

  **Note:** Since pyglet 1.1

  **Return type** *float*

  **Returns** The next timestamp, or *None* if there are no more video frames.
StaticMemorySource.play()

Play the source.

This is a convenience method which creates a Player for this source and plays it immediately.

Return type Player

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.

Constructor:

__init__(source)

Construct a StaticSource for the data in source.

Parameters source (Source) – The source to read and decode audio and video data from.

Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_animation()</td>
<td>Import all video frames into memory as an Animation.</td>
</tr>
<tr>
<td>get_audio_data(bytes)</td>
<td>Get the next video frame.</td>
</tr>
<tr>
<td>get_next_video_frame()</td>
<td>Get the next video frame.</td>
</tr>
<tr>
<td>get_next_video_timestamp()</td>
<td>Get the timestamp of the next video frame.</td>
</tr>
<tr>
<td>play()</td>
<td>Play the source.</td>
</tr>
<tr>
<td>seek(timestamp)</td>
<td>Seek to given timestamp.</td>
</tr>
</tbody>
</table>

Attributes:

audio_format

Continued on next page
Methods

StaticSource.get_audio_data(bytes)

Inherited members

Methods

StaticSource.get_animation()

Import all video frames into memory as an Animation.

An empty animation will be returned if the source has no video. Otherwise, the animation will contain all unplayed video frames (the entire source, if it has not been queued on a player). After creating the animation, the source will be at EOS.

This method is unsuitable for videos running longer than a few seconds.

Note: Since pyglet 1.1

Return type pyglet.image.Animation

StaticSource.get_next_video_frame()

Get the next video frame.

Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

Note: Since pyglet 1.1

Return type pyglet.image.AbstractImage

Returns The next video frame image, or None if the video frame could not be decoded or there are no more video frames.

StaticSource.get_next_video_timestamp()

Get the timestamp of the next video frame.

Note: Since pyglet 1.1

Return type float

Returns The next timestamp, or None if there are no more video frames.

StaticSource.play()

Play the source.

This is a convenience method which creates a Player for this source and plays it immediately.
Return type  *Player*

```python
StaticSource.seek(timestamp)
```

Seek to given timestamp.

Attributes

```python
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.

Methods:

```
get_animation()  Import all video frames into memory as an Animation.
get_audio_data(bytes)  Get next packet of audio data.
get_next_video_frame()  Get the next video frame.
get_next_video_timestamp()  Get the timestamp of the next video frame.
play()  Play the source.
seek(timestamp)  Seek to given timestamp.
```

Attributes:

```
audio_format
duration  The length of the source, in seconds.
info
is_queued  Determine if this source has been queued on a Player yet.
video_format
```
Attributes

`StreamingSource.is_queued`
Determine if this source has been queued on a `Player` yet.
Read-only.
Type: `bool`

Inherited members

Methods

`StreamingSource.get_animation()`
Import all video frames into memory as an `Animation`.
An empty animation will be returned if the source has no video. Otherwise, the animation will contain all unplayed video frames (the entire source, if it has not been queued on a player). After creating the animation, the source will be at EOS.
This method is unsuitable for videos running longer than a few seconds.

Note: Since pyglet 1.1

Return type: `pyglet.image.Animation`

`StreamingSource.get_audio_data(bytes)`
Get next packet of audio data.
Parameters: `bytes (int)` – Maximum number of bytes of data to return.
Return type: `AudioData`
Returns: Next packet of audio data, or None if there is no (more) data.

`StreamingSource.get_next_video_frame()`
Get the next video frame.
Video frames may share memory: the previous frame may be invalidated or corrupted when this method is called unless the application has made a copy of it.

Note: Since pyglet 1.1

Return type: `pyglet.image.AbstractImage`

Returns: The next video frame image, or None if the video frame could not be decoded or there are no more video frames.

`StreamingSource.get_next_video_timestamp()`
Get the timestamp of the next video frame.

Note: Since pyglet 1.1

Return type: `float`

Returns: The next timestamp, or None if there are no more video frames.
StreamingSource.play()
    Play the source.
    This is a convenience method which creates a Player for this source and plays it immediately.

    Return type  Player

StreamingSource.seek(timestamp)
    Seek to given timestamp.

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

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.

    Constructor:
    __init__ (width, height, sample_aspect=1.0)
WorkerThread Class

class WorkerThread (target=None)

Constructor:
__init__(target=None)

Methods:

- clear_jobs()
- get_job()
- notify() Interrupt the current sleep operation.
- put_job(job)
- run()
- sleep(timeout) Wait for some amount of time, or until notified.
- start()
- stop() Stop the thread and wait for it to terminate.

Methods
WorkerThread.clear_jobs()
WorkerThread.get_job()
WorkerThread.put_job(job)
WorkerThread.run()

Inherited members

Methods

WorkerThread.notify() Interrupt the current sleep operation.
If the thread is currently sleeping, it will be woken immediately, instead of waiting the full duration of the timeout.

WorkerThread.sleep(timeout) Wait for some amount of time, or until notified.

Parameters timeout (float) – Time to wait, in seconds.

WorkerThread.start()
WorkerThread.stop() Stop the thread and wait for it to terminate.
The `stop` instance variable is set to `True` and the condition is notified. It is the responsibility of the `run` method to check the value of `stop` after each sleep or wait and to return if set.

### Exceptions

<table>
<thead>
<tr>
<th>CannotSeekException</th>
<th>MediaException</th>
<th>MediaFormatException</th>
</tr>
</thead>
</table>

- **CannotSeekException**
  - Exception defined in `pyglet.media`
  - `exception CannotSeekException`

- **MediaException**
  - Exception defined in `pyglet.media`
  - `exception MediaException`

- **MediaFormatException**
  - Exception defined in `pyglet.media`
  - `exception MediaFormatException`

### Functions
get_audio_driver()     Defined in pyglet.media
get_silent_audio_driver()     Defined in pyglet.media
get_source_loader()     Defined in pyglet.media
load(filename[, file, streaming])     Load a source from a file.

get_audio_driver Function     Defined in pyglet.media
get_audio_driver()

get_silent_audio_driver Function     Defined in pyglet.media
get_silent_audio_driver()

get_source_loader Function     Defined in pyglet.media
get_source_loader()

load Function     Defined in pyglet.media
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

have_avbin = 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.

listener = <pyglet.media._LegacyListener object>
The singleton AbstractListener object.

Warning:  Deprecated. Use AbstractAudioDriver.get_listener

Type  AbstractListener
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.levell', followed
# by the 'res/images' directory.
path = ['@levels.levell', '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

<table>
<thead>
<tr>
<th>Location Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileLocation</td>
<td>Location on the filesystem.</td>
</tr>
<tr>
<td>Loader</td>
<td>Load program resource files from disk.</td>
</tr>
<tr>
<td>Location</td>
<td>Abstract resource location.</td>
</tr>
<tr>
<td>URLLocation</td>
<td>Location on the network.</td>
</tr>
<tr>
<td>ZIPLocation</td>
<td>Location within a ZIP file.</td>
</tr>
</tbody>
</table>

FileLocation Class

class FileLocation (path)
   Location on the filesystem.
   Constructor:
   __init__ (path)
      Create a location given a relative or absolute path.
   Parameters path (str) – Path on the filesystem.
   Methods:
   open(filename[, mode])

Methods
FileLocation.open (filename, mode='rb')

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.

Constructor:

```python
__init__(path=None, script_home=None)
```
Create a loader for the given path.

If no path is specified it defaults to `['.']`; that is, just the program directory.

See the module documentation for details on the path format.

Parameters

- **path** ([list of str]) – List of locations to search for resources.
- **script_home** (str) – Base location of relative files. Defaults to the result of `get_script_home`.

Methods:

- `add_font(name)` Add a font resource to the application.
- `animation(name[, flip_x, flip_y, rotate])` Load an animation with optional transformation.
- `attributed(name)` Load an attributed text document.
- `file(name[, mode])` Load a resource.
- `get_cached_animation_names()` Get a list of animation filenames that have been cached.
- `get_cached_image_names()` Get a list of image filenames that have been cached.
- `get_cached_texture_names()` Get the names of textures currently cached.
- `get_texture_bins()` Get a list of texture bins in use.
- `html(name)` Load an HTML document.
- `image(name[, flip_x, flip_y, rotate, atlas])` Load an image with optional transformation.
- `location(name)` Get the location of a resource.
- `media(name[, streaming])` Load a sound or video resource.
- `reindex()` Refresh the file index.
- `text(name)` Load a plain text document.
- `texture(name)` Load a texture.

Methods

**Loader**. `add_font(name)` 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:

```python
resource.add_font('action_man.ttf')
action_man = font.load('Action Man')
```

Parameters **name** (str) – Filename of the font resource to add.

**Loader**. `animation(name[, flip_x=False, flip_y=False, rotate=0])` 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*

**Loader.attributed** *(name)*  
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*

**Loader.file** *(name, mode='rb')*  
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*

**Loader.get_cached_animation_names()**  
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*

**Loader.get_cached_image_names()**  
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*

**Loader.get_cached_texture_names()**  
Get the names of textures currently cached.

    **Return type**  
    *list of str*

**Loader.get_texture_bins()**  
Get a list of texture bins in use.

    This is useful for debugging and profiling only.

    **Return type**  
    *list*

    **Returns**  
    *List of TextureBin*

**Loader.html** *(name)*  
Load an HTML document.

    **Parameters**  
    • **name** *(str)* – Filename of the HTML resource to load.
Loader.image(name, flip_x=False, flip_y=False, rotate=0, atlas=True)

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  *FormattedDocument*

Returns  A complete texture if the image is large or not in an atlas, otherwise a `TextureRegion` of a texture atlas.

Loader.location(name)

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*

Loader.media(name, streaming=True)

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*

Loader.reindex()

Refresh the file index.

You must call this method if `path` is changed or the filesystem layout changes.

Loader.text(name)

Load a plain text document.

Parameters  **name** (`str`) – Filename of the plain text resource to load.

Return type  *UnformattedDocument*
Loader.texture(name)
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

---

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.

Methods:

open(filename[, mode]) Open a file at this location.

---

Methods

Location.open(filename, mode='rb')
Open a file at this location.

Parameters

- filename (str) – The filename to open. Absolute paths are not supported. Relative paths are not supported by most locations (you should specify only a filename with no path component).

- mode (str) – The file mode to open with. Only files opened on the filesystem make use of this parameter; others ignore it.

Return type file object

---

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.

Constructor:

_init_(base_url)
Create a location given a base URL.

Parameters base_url (str) – URL string to prepend to filenames.

Methods:

open(filename[, mode])

Methods

URLLocation.open (filename, mode='rb')

ZIPLocation Class

class ZIPLocation (zip, dir)
Location within a ZIP file.

Constructor:

_init_(zip, dir)
Create a location given an open ZIP file and a path within that file.

Parameters

• zip (zipfile.ZipFile) – An open ZIP file from the zipfile module.

• dir (str) – A path within that ZIP file. Can be empty to specify files at the top level of the ZIP file.

Methods:

open(filename[, mode])

Methods

ZIPLocation.open (filename, mode='rb')
Exceptions

- `ResourceNotFoundException` (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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_script_home()</code></td>
<td>Get the directory containing the program entry module.</td>
</tr>
<tr>
<td><code>get_settings_path(name)</code></td>
<td>Get a directory to save user preferences.</td>
</tr>
</tbody>
</table>

`get_script_home` Function Defined in `pyglet.resource`

```
ge_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 workaround 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 _DefaultLoader.add_font of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>`
  
  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:

  ```python
  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 _DefaultLoader.animation of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>`

  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 _DefaultLoader.attributed of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>`

  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 _DefaultLoader.file of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>`

  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 _DefaultLoader.get_cached_animation_names of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>`

  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 _DefaultLoader.get_cached_image_names of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>
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 _DefaultLoader.get_cached_texture_names of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>
Get the names of textures currently cached.

Return type list of str

get_texture_bins = <bound method _DefaultLoader.get_texture_bins of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>
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 _DefaultLoader.html of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>
Load an HTML document.

Parameters name (str) – Filename of the HTML resource to load.

Return type FormattedDocument

image = <bound method _DefaultLoader.image of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>
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 _DefaultLoader.location of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>
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 _DefaultLoader.media of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>
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 _DefaultLoader.reindex of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>
Refresh the file index.

You must call this method if path is changed or the filesystem layout changes.

text = <bound method _DefaultLoader.text of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>
Load a plain text document.

Parameters name (str) – Filename of the plain text resource to load.

Return type UnformattedDocument

texture = <bound method _DefaultLoader.texture of <pyglet.resource._DefaultLoader object at 0x7f90fc1bc650>>
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
- sys
- weakref
- zipfile
**pyglet**

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:

```python
import pyglet

ball_image = pyglet.image.load('ball.png')
bball = 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 \((\text{anchor}_x, \text{anchor}_y)\).

**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:

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

ballSprites = []
for i in range(100):
    x, y = i * 10, 50
    ballSprites.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

<table>
<thead>
<tr>
<th>Sprite</th>
<th>Instance of an on-screen image.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpriteGroup</td>
<td>Shared sprite rendering group.</td>
</tr>
</tbody>
</table>

```python
pyglet.event.EventDispatcher — pyglet.sprite.Sprite
```

**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.

**Constructor:**

```python
def __init__(img, x=0, y=0, blend_src=770, blend_dest=771, batch=None, group=None, usage='dynamic', subpixel=False)
```

Create a sprite.

**Parameters**

- `img` *(AbstractImage or Animation)* – Image or animation to display.
- `x` *(int)* – X coordinate of the sprite.
- `y` *(int)* – Y coordinate of the sprite.
- `blend_src` *(int)* – OpenGL blend source mode. The default is suitable for compositing sprites drawn from back-to-front.
- `blend_dest` *(int)* – OpenGL blend destination mode. The default is suitable for compositing sprites drawn from back-to-front.
- `batch` *(Batch)* – Optional batch to add the sprite to.
- `group` *(Group)* – Optional parent group of the sprite.
- `usage` *(str)* – Vertex buffer object usage hint, one of "none", "stream", "dynamic" (default) or "static". Applies only to vertex data.
- `subpixel` *(bool)* – Allow floating-point coordinates for the sprite. By default, coordinates are restricted to integer values.

**Methods:**

- `delete()` Force immediate removal of the sprite from video memory.
- `draw()` Draw the sprite at its current position.
- `set_position(x, y)` Set the X and Y coordinates of the sprite simultaneously.

**Events:**
**Attributes:**

- **batch**  
  Graphics batch.
- **color**  
  Blend color.
- **event_types**  
  Parent graphics group.
- **height**  
  Scaled height of the sprite.
- **image**  
  Image or animation to display.
- **opacity**  
  Blend opacity.
- **position**  
  The (x, y) coordinates of the sprite.
- **rotation**  
  Clockwise rotation of the sprite, in degrees.
- **scale**  
  Scaling factor.
- **visible**
- **width**  
  Scaled width of the sprite.
- **x**  
  X coordinate of the sprite.
- **y**  
  Y coordinate of the sprite.

**Methods**

- **Sprite.delete()**  
  Force immediate removal of the sprite from video memory.
  
  This is often necessary when using batches, as the Python garbage collector will not necessarily call the finalizer as soon as the sprite is garbage.

- **Sprite.draw()**  
  Draw the sprite at its current position.
  
  See the module documentation for hints on drawing multiple sprites efficiently.

- **Sprite.set_position(x, y)**  
  Set the X and Y coordinates of the sprite simultaneously.

  **Parameters**

  - **x (int)** – X coordinate of the sprite.
  - **y (int)** – Y coordinate of the sprite.

**Events**

- **Sprite.on_animation_end()**  
  The sprite animation reached the final frame.
  
  The event is triggered only if the sprite has an animation, not an image. For looping animations, the event is triggered each time the animation loops.

**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.

**Type** *(int, int)*

**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.visible**
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.dispatch_event(event_type, *args)
Dispatch a single event to the attached handlers.

The event is propagated to all handlers from from the top of the stack until one returns EVENT_HANDLED. This method should be used only by EventDispatcher implementors; applications should call the dispatch_events method.

Since pyglet 1.2, the method returns EVENT_HANDLED if an event handler returned EVENT_HANDLED or EVENT_UNHANDLED if one or more event handlers were invoked but returned only EVENT_UNHANDLED; otherwise False. In pyglet 1.1 and earlier, the return value is always None.

Parameters

• event_type (str) – Name of the event.
• args (sequence) – Arguments to pass to the event handler.

Return type bool or None

Returns (Since pyglet 1.2) EVENT_HANDLED if an event handler returned EVENT_HANDLED; EVENT_UNHANDLED if one or more event handlers were invoked but returned only EVENT_UNHANDLED; otherwise False. In pyglet 1.1 and earlier, the return value is always None.

Sprite.event(*args)
Function decorator for an event handler.

Usage:

```python
win = window.Window()

@win.event
def on_resize(self, width, height):
    # ...
```

or:

```python
@win.event('on_resize')
def foo(self, width, height):
    # ...
```
Sprite.pop_handlers()
    Pop the top level of event handlers off the stack.

Sprite.push_handlers(*args, **kwargs)
    Push a level onto the top of the handler stack, then attach zero or more event handlers.
    If keyword arguments are given, they name the event type to attach. Otherwise, a callable’s
    __name__ attribute will be used. Any other object may also be specified, in which case it will
    be searched for callables with event names.

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.

Sprite.remove_handler(name, handler)
    Remove a single event handler.
    The given event handler is removed from the first handler stack frame it appears in. The handler
    must be the exact same callable as passed to set_handler, set_handlers or push_handlers; and the
    name must match the event type it is bound to.
    No error is raised if the event handler is not set.
    
    Parameters
    • name (str) – Name of the event type to remove.
    • handler (callable) – Event handler to remove.

Sprite.remove_handlers(*args, **kwargs)
    Remove event handlers from the event stack.
    See push_handlers for the accepted argument types. All handlers are removed from the first stack
    frame that contains any of the given handlers. No error is raised if any handler does not appear in
    that frame, or if no stack frame contains any of the given handlers.
    If the stack frame is empty after removing the handlers, it is removed from the stack. Note that this
    interferes with the expected symmetry of push_handlers and pop_handlers.

Sprite.set_handler(name, handler)
    Attach a single event handler.
    
    Parameters
    • name (str) – Name of the event type to attach to.
    • handler (callable) – Event handler to attach.

Sprite.set_handlers(*args, **kwargs)
    Attach one or more event handlers to the top level of the handler stack.
    See push_handlers for the accepted argument types.
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.

Constructor:

__init__(texture, blend_src, blend_dest, parent=None)

Create a sprite group.

The group is created internally within Sprite; applications usually do not need to explicitly create it.

Parameters

• texture (Texture) – The (top-level) texture containing the sprite image.

• blend_src (int) – OpenGL blend source mode; for example, GL_SRC_ALPHA.

• blend_dest (int) – OpenGL blend destination mode; for example, GL_ONE_MINUS_SRC_ALPHA.

• parent (Group) – Optional parent group.

Methods:

set_state()

unset_state()

Methods

SpriteGroup.set_state()

SpriteGroup.unset_state()

Inherited members

Methods

SpriteGroup.set_state_recursive()

Set this group and its ancestry.

Call this method if you are using a group in isolation: the parent groups will be called in top-down order, with this class’s set being called last.

SpriteGroup.unset_state_recursive()

Unset this group and its ancestry.

The inverse of set_state_recursive.
Variables

`compat_platform = 'linux2'`

```python
str(object='') -> string
```

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

Notes

Defined

- clock
- event
- gl
- glext_arb
- glu
- graphics
- image
- lib
- lib_glx
- math
- sys

**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:

```python
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:

```python
label = pyglet.text.HTMLLabel('<b>Hello</b>, <i>world</i>',
                             x=10, y=10)
```

Either label can then be drawn at any time with:

```python
label.draw()
```

For details on the subset of HTML supported, see [pyglet.text.formats.html](http://pyglet.readthedocs.io/en/1.2.4/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

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>caret</td>
<td>Provides keyboard and mouse editing procedures for text layout.</td>
</tr>
<tr>
<td>document</td>
<td>Formatted and unformatted document interfaces used by text layout.</td>
</tr>
<tr>
<td>formats</td>
<td>Document formats.</td>
</tr>
<tr>
<td>layout</td>
<td>Render simple text and formatted documents efficiently.</td>
</tr>
<tr>
<td>runlist</td>
<td>Run list encoding utilities.</td>
</tr>
</tbody>
</table>

**pyglet.text.caret**  Provides keyboard and mouse editing procedures for text layout.

Example usage:

```python
define usage:
from pyglet import window
from pyglet.text import layout, caret

my_window = window.Window(...)
my_layout = layout.IncrementalTextLayout(...)  # Change layout to IncrementalTextLayout
my_caret = caret.Caret(my_layout)
my_window.push_handlers(my_caret)
```

**Note:** Since pyglet 1.1

---

**Caret**  Visible text insertion marker for `pyglet.text.layout.IncrementalTextLayout`.

### Classes

**Caret Class**

```python
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.

**Constructor:**

```python
__init__(layout, batch=None, color=(0, 0, 0))
```

Create a caret for a layout.

By default the layout’s batch is used, so the caret does not need to be drawn explicitly.

**Parameters**

- `layout (TextLayout)` – Layout to control.
- `batch (Batch)` – Graphics batch to add vertices to.
- `color ((int, int, int))` – RGB tuple with components in range [0, 255].

**Methods:**

- `delete()` Remove the caret from its batch.
- `get_style(attribute)` Get the document’s named style at the caret’s current position.
- `move_to_point(x, y)` Move the caret close to the given window coordinate.
- `on_activate()` Handler for the `pyglet.window.Window.on_activate` event.
- `on_deactivate()` Handler for the `pyglet.window.Window.on_deactivate` event.
- `on_layout_update()`
- `on_mouse_drag(x, y, dx, dy, buttons, modifiers)` Handler for the `pyglet.window.Window.on_mouse_drag` event.
- `on_mouse_press(x, y, button, modifiers)` Handler for the `pyglet.window.Window.on_mouse_press` event.
- `on_mouse_scroll(x, y, scroll_x, scroll_y)` Handler for the `pyglet.window.Window.on_mouse_scroll` event.
- `on_text(text)` Handler for the `pyglet.window.Window.on_text` event.
- `on_text_motion(motion[, select])` Handler for the `pyglet.window.Window.on_text_motion` event.
- `on_text_motion_select(motion)` Handler for the `pyglet.window.Window.on_text_motion_select` event.
- `select_paragraph(x, y)` Select the paragraph at the given window coordinate.
- `select_to_point(x, y)` Move the caret close to the given window coordinate while maintaining the mark.
- `select_word(x, y)` Select the word at the given window coordinate.
- `set_style(attributes)` Set the document style at the caret’s current position.

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERIOD</td>
<td>Blink period, in seconds.</td>
</tr>
<tr>
<td>SCROLL_INCREMENT</td>
<td>Pixels to scroll viewport per mouse scroll wheel movement.</td>
</tr>
<tr>
<td>color</td>
<td>Caret color.</td>
</tr>
<tr>
<td>line</td>
<td>Index of line containing the caret’s position.</td>
</tr>
<tr>
<td>mark</td>
<td>Position of immovable end of text selection within document.</td>
</tr>
<tr>
<td>position</td>
<td>Position of caret within document.</td>
</tr>
<tr>
<td>visible</td>
<td>Caret visibility.</td>
</tr>
</tbody>
</table>

**Methods**

Caret . `delete()` Remove the caret from its batch.
Also disconnects the caret from further layout events.

Caret.

get_style(attribute)

Get the document’s named style at the caret’s current position.

If there is a text selection and the style varies over the selection, `pyglet.text.document.STYLE_INDETERMINATE` is returned.

Parameters attribute (str) – Name of style attribute to retrieve. See `pyglet.text.document` for a list of recognised attribute names.

Return type object

Caret.

move_to_point(x, y)

Move the caret close to the given window coordinate.

The mark will be reset to None.

Parameters

• x (int) – X coordinate.
• y (int) – Y coordinate.

Caret.

on_activate()

Handler for the `pyglet.window.Window.on_activate` event.

The caret is hidden when the window is not active.

Caret.

on_deactivate()

Handler for the `pyglet.window.Window.on_deactivate` event.

The caret is hidden when the window is not active.

Caret.

on_layout_update()

Caret.

on_mouse_drag(x, y, dx, dy, buttons, modifiers)

Handler for the `pyglet.window.Window.on_mouse_drag` event.

Mouse handlers do not check the bounds of the coordinates: GUI toolkits should filter events that do not intersect
the layout before invoking this handler.

Caret.

on_mouse_press(x, y, button, modifiers)

Handler for the `pyglet.window.Window.on_mouse_press` event.

Mouse handlers do not check the bounds of the coordinates: GUI toolkits should filter events that do not intersect the layout before invoking this handler.

This handler keeps track of the number of mouse presses within a short span of time and uses this to reconstruct double- and triple-click events for selecting words and paragraphs. This technique is not suitable when a GUI toolkit is in use, as the active widget must also be tracked. Do not use this mouse handler if a GUI toolkit is being used.

Caret.

on_mouse_scroll(x, y, scroll_x, scroll_y)

Handler for the `pyglet.window.Window.on_mouse_scroll` event.

Mouse handlers do not check the bounds of the coordinates: GUI toolkits should filter events that do not intersect the layout before invoking this handler.

The layout viewport is scrolled by `SCROLL_INCREMENT` pixels per “click”.

Caret.

on_text(text)

Handler for the `pyglet.window.Window.on_text` event.

Caret keyboard handlers assume the layout always has keyboard focus. GUI toolkits should filter keyboard and text events by widget focus before invoking this handler.
Caret.on_text_motion(motion, select=False)
Handler for the pyglet.window.Window.on_text_motion event.

Caret keyboard handlers assume the layout always has keyboard focus. GUI toolkits should filter keyboard and
text events by widget focus before invoking this handler.

Caret.on_text_motion_select(motion)
Handler for the pyglet.window.Window.on_text_motion_select event.

Caret keyboard handlers assume the layout always has keyboard focus. GUI toolkits should filter keyboard and
text events by widget focus before invoking this handler.

Caret.select_paragraph(x, y)
Select the paragraph at the given window coordinate.

Parameters

• x (int) – X coordinate.
• y (int) – Y coordinate.

Caret.select_to_point(x, y)
Move the caret close to the given window coordinate while maintaining the mark.

Parameters

• x (int) – X coordinate.
• y (int) – Y coordinate.

Caret.select_word(x, y)
Select the word at the given window coordinate.

Parameters

• x (int) – X coordinate.
• y (int) – Y coordinate.

Caret.set_style(attributes)
Set the document style at the caret’s current position.

If there is a text selection the style is modified immediately. Otherwise, the next text that is entered before the
position is modified will take on the given style.

Parameters attributes (dict) – Dict mapping attribute names to style values. See pyglet.text.document for a list of recognised attribute names.

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, \textit{position} is modified to place the caret on requested line while maintaining the closest possible X offset.

\textbf{Type} int

\texttt{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 \texttt{None} when there is no selection.

\textbf{Type} int

\texttt{Caret.position}

Position of caret within document.

\textbf{Type} int

\texttt{Caret.visible}

Caret visibility.

The caret may be hidden despite this property due to the periodic blinking or by \texttt{on_deactivate} if the event handler is attached to a window.

\textbf{Type} bool

\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Defined} & \textbf{Notes} \\
\hline
\texttt{clock} & \texttt{event} & \texttt{key} & \texttt{re} & \texttt{time} \\
\hline
\end{tabular}

\textbf{pyglet.text.document} Formatted and unformatted document interfaces used by text layout.

\textbf{Abstract representation} Styled text in pyglet is represented by one of the \textit{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:

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|}
\hline
0 & 5 & 10 & 15 & 20 \\
\hline
The cat sat on the mat. \\
+++++++ & ++ & ++++++ & \textit{"bold"} & \texttt{"italic"} \\
+++++++ & ++ & \\
\hline
\end{tabular}
\end{center}

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 \texttt{"bold"} over ranges (0-7, 15-22) and \texttt{"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 \texttt{"bold"} or \texttt{"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](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 unstructued 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.
- **kerning** 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 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
- **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

---

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractDocument</td>
<td>Abstract document interface used by all <code>pyglet.text</code> classes.</td>
</tr>
<tr>
<td>FormattedDocument</td>
<td>Simple implementation of a document that maintains text formatting.</td>
</tr>
<tr>
<td>InlineElement</td>
<td>Arbitrary inline element positioned within a formatted document.</td>
</tr>
<tr>
<td>UnformattedDocument</td>
<td>A document having uniform style over all text.</td>
</tr>
</tbody>
</table>

**Classes**

```
```

**AbstractDocument** Class

class AbstractDocument

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`.

**Constructor:**

```
__init__(text='')
```

**Methods:**

```
delete_text(start, end) Delete text from the document.
get_element(position) Get the element at a specified position.
get_font(position[, dpi]) Get the font instance used at the given position.
get_font_runs([dpi]) Get a style iterator over the `pyglet.font.Font` instances used in the document.
get_paragraph_end(pos) Get the end position of a paragraph.
get_paragraph_start(pos) Get the starting position of a paragraph.
get_style(attribute[, position]) Get an attribute style at the given position.
get_style_range(attribute, start, end) Get an attribute style over the given range.
get_style_runs(attribute) Get a style iterator over the given style attribute.
inset_element(position, element[, attributes]) Insert a element into the document.
```
Table 2.253 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert_text(start, text[, attributes])</code></td>
<td>Insert text into the document.</td>
</tr>
<tr>
<td><code>set_paragraph_style(start, end, attributes)</code></td>
<td>Set the style for a range of paragraphs.</td>
</tr>
<tr>
<td><code>set_style(start, end, attributes)</code></td>
<td>Set text style of some or all of the document.</td>
</tr>
</tbody>
</table>

Events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on_delete_text(start, end)</code></td>
<td>Text was deleted from the document.</td>
</tr>
<tr>
<td><code>on_insert_text(start, text)</code></td>
<td>Text was inserted into the document.</td>
</tr>
<tr>
<td><code>on_style_text(start, end, attributes)</code></td>
<td>Text character style was modified.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>event_types</code></td>
<td></td>
</tr>
<tr>
<td><code>text</code></td>
<td>Document text.</td>
</tr>
</tbody>
</table>

Methods

`AbstractDocument.delete_text(start, end)` Delete text from the document.

Parameters

- `start` (int) – Starting character position to delete from.
- `end` (int) – Ending character position to delete to (exclusive).

`AbstractDocument.get_element(position)` Get the element at a specified position.

Parameters `position` (int) – Position in the document of the element.

Return type `InlineElement`

`AbstractDocument.get_font(position, dpi=None)` Get the font instance used at the given position.

See `get_font_runs`

Parameters

- `position` (int) – Character position of document to query.
- `dpi` (float) – Optional resolution to construct fonts at. See `pyglet.font.load`.

Return type `pyglet.font.Font`

Returns The font at the given position.

`AbstractDocument.get_font_runs(dpi=None)` Get a style iterator over the `pyglet.font.Font` instances used in the document.

The font instances are created on-demand by inspection of the `font_name`, `font_size`, `bold` and `italic` style attributes.

Parameters `dpi` (float) – Optional resolution to construct fonts at. See `pyglet.font.load`. 
Return type: AbstractRunIterator

AbstractDocument.get_paragraph_end(pos)
Get the end position of a paragraph.

Parameters
pos (int) – Character position within paragraph.

Return type: int

AbstractDocument.get_paragraph_start(pos)
Get the starting position of a paragraph.

Parameters
pos (int) – Character position within paragraph.

Return type: int

AbstractDocument.get_style(attribute, position=0)
Get an attribute style at the given position.

Parameters

• attribute (str) – Name of style attribute to query.

• position (int) – Character position of document to query.

Returns The style set for the attribute at the given position.

AbstractDocument.get_style_range(attribute, start, end)
Get an attribute style over the given range.

If the style varies over the range, STYLE_INDETERMINATE is returned.

Parameters

• attribute (str) – Name of style attribute to query.

• start (int) – Starting character position.

• end (int) – Ending character position (exclusive).

Returns The style set for the attribute over the given range, or STYLE_INDETERMINATE if more than one value is set.

AbstractDocument.get_style_runs(attribute)
Get a style iterator over the given style attribute.

Parameters
attribute (str) – Name of style attribute to query.

Return type: AbstractRunIterator

AbstractDocument.insert_element(position, element, attributes=None)
Insert a element into the document.

See the InlineElement class documentation for details of usage.

Parameters

• position (int) – Character insertion point within document.

• element (InlineElement) – Element to insert.

• attributes (dict) – Optional dictionary giving named style attributes of the inserted text.

AbstractDocument.insert_text(start, text, attributes=None)
Insert text into the document.

Parameters

• start (int) – Character insertion point within document.
• **text** (*str*) – Text to insert.
• **attributes** (*dict*) – Optional dictionary giving named style attributes of the inserted text.

AbstractDocument.**set_paragraph_style**(start, end, attributes)
Set the style for a range of paragraphs.

This is a convenience method for `set_style` that aligns the character range to the enclosing paragraph(s).

**Parameters**

• **start** (*int*) – Starting character position.
• **end** (*int*) – Ending character position (exclusive).
• **attributes** (*dict*) – Dictionary giving named style attributes of the paragraphs.

AbstractDocument.**set_style**(start, end, attributes)
Set text style of some or all of the document.

**Parameters**

• **start** (*int*) – Starting character position.
• **end** (*int*) – Ending character position (exclusive).
• **attributes** (*dict*) – Dictionary giving named style attributes of the text.

**Events**

AbstractDocument.**on_delete_text**(start, end)
Text was deleted from the document.

**Parameters**

• **start** (*int*) – Starting character position of deleted text.
• **end** (*int*) – Ending character position of deleted text (exclusive).

AbstractDocument.**on_insert_text**(start, text)
Text was inserted into the document.

**Parameters**

• **start** (*int*) – Character insertion point within document.
• **text** (*str*) – The text that was inserted.

AbstractDocument.**on_style_text**(start, end, attributes)
Text character style was modified.

**Parameters**

• **start** (*int*) – Starting character position of modified text.
• **end** (*int*) – Ending character position of modified text (exclusive).
• **attributes** (*dict*) – Dictionary giving updated named style attributes of the 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.dispatch_event(event_type, *args)
Dispatch a single event to the attached handlers.

The event is propagated to all handlers from from the top of the stack until one returns EVENT_HANDLED. This method should be used only by EventDispatcher implementors; applications should call the dispatch_events method.

Since pyglet 1.2, the method returns EVENT_HANDLED if an event handler returned EVENT_HANDLED or EVENT_UNHANDLED if all events returned EVENT_UNHANDLED. If no matching event handlers are in the stack, False is returned.

Parameters

- **event_type** (str) – Name of the event.
- **args** (sequence) – Arguments to pass to the event handler.

Return type bool or None

Returns (Since pyglet 1.2) EVENT_HANDLED if an event handler returned EVENT_HANDLED; EVENT_UNHANDLED if one or more event handlers were invoked but returned only EVENT_UNHANDLED; otherwise False. In pyglet 1.1 and earlier, the return value is always None.

AbstractDocument.event(*args)
Function decorator for an event handler.

Usage:

```python
win = window.Window()

@win.event
def on_resize(self, width, height):
    # ...
```

or:

```python
@win.event('on_resize')
def foo(self, width, height):
    # ...
```

AbstractDocument.pop_handlers()
Pop the top level of event handlers off the stack.

AbstractDocument.push_handlers(*args, **kwargs)
Push a level onto the top of the handler stack, then attach zero or more event handlers.

If keyword arguments are given, they name the event type to attach. Otherwise, a callable’s __name__ attribute will be used. Any other object may also be specified, in which case it will be searched for callables with event names.

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.
AbstractDocument.remove_handler(name, handler)

Remove a single event handler.

The given event handler is removed from the first handler stack frame it appears in. The handler must be the exact same callable as passed to set_handler, set_handlers or push_handlers; and the name must match the event type it is bound to.

No error is raised if the event handler is not set.

Parameters

- name (str) – Name of the event type to remove.
- handler (callable) – Event handler to remove.

AbstractDocument.remove_handlers(*args, **kwargs)

Remove event handlers from the event stack.

See push_handlers for the accepted argument types. All handlers are removed from the first stack frame that contains any of the given handlers. No error is raised if any handler does not appear in that frame, or if no stack frame contains any of the given handlers.

If the stack frame is empty after removing the handlers, it is removed from the stack. Note that this interferes with the expected symmetry of push_handlers and pop_handlers.

AbstractDocument.set_handler(name, handler)

Attach a single event handler.

Parameters

- name (str) – Name of the event type to attach to.
- handler (callable) – Event handler to attach.

AbstractDocument.set_handlers(*args, **kwargs)

Attach one or more event handlers to the top level of the handler stack.

See push_handlers for the accepted argument types.

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.

Constructor:

__init__(text= '')

Methods:

- delete_text(start, end) – Delete text from the document.
- get_element(position) – Get the element at a specified position.
- get_element_runs() –
- get_font(position[, dpi]) –
- get_font_runs([dpi]) –

Continued on next page
Table 2.256 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_paragraph_end(pos)</td>
<td>Get the end position of a paragraph.</td>
</tr>
<tr>
<td>get_paragraph_start(pos)</td>
<td>Get the starting position of a paragraph.</td>
</tr>
<tr>
<td>get_style(attribute[, position])</td>
<td>Get an attribute style over the given range.</td>
</tr>
<tr>
<td>get_style_range(attribute, start, end)</td>
<td>Get an attribute style over the given range.</td>
</tr>
<tr>
<td>get_style_runs(attribute)</td>
<td>Insert a element into the document.</td>
</tr>
<tr>
<td>insert_element(position, element[, attributes])</td>
<td>Insert text into the document.</td>
</tr>
<tr>
<td>insert_text(start, text[, attributes])</td>
<td>Insert text into the document.</td>
</tr>
<tr>
<td>set_paragraph_style(start, end, attributes)</td>
<td>Set the style for a range of paragraphs.</td>
</tr>
<tr>
<td>set_style(start, end, attributes)</td>
<td>Set text style of some or all of the document.</td>
</tr>
</tbody>
</table>

Events:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>on_delete_text(start, end)</td>
<td>Text was deleted from the document.</td>
</tr>
<tr>
<td>on_insert_text(start, text)</td>
<td>Text was inserted into the document.</td>
</tr>
<tr>
<td>on_style_text(start, end, attributes)</td>
<td>Text character style was modified.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_types</td>
<td>Document text.</td>
</tr>
<tr>
<td>text</td>
<td>Document text.</td>
</tr>
</tbody>
</table>

Methods

FormattedDocument.get_element_runs()

FormattedDocument.get_font(position, dpi=None)

FormattedDocument.get_font_runs(dpi=None)

FormattedDocument.get_style(attribute, position=0)

FormattedDocument.get_style_runs(attribute)

Inherited members

Methods

FormattedDocument.delete_text(start, end)

Delete text from the document.

Parameters

- start (int) – Starting character position to delete from.
- end (int) – Ending character position to delete to (exclusive).

FormattedDocument.dispatch_event(event_type, *args)

Dispatch a single event to the attached handlers.
The event is propagated to all handlers from from the top of the stack until one returns \texttt{EVENT\_HANDLED}. This method should be used only by \texttt{EventDispatcher} implementors; applications should call the \texttt{dispatch\_events} method.

Since pyglet 1.2, the method returns \texttt{EVENT\_HANDLED} if an event handler returned \texttt{EVENT\_HANDLED} or \texttt{EVENT\_UNHANDLED} if all events returned \texttt{EVENT\_UNHANDLED}. If no matching event handlers are in the stack, \texttt{False} is returned.

\begin{itemize}
  \item \texttt{event\_type} \texttt{(str)} – Name of the event.
  \item \texttt{args} \texttt{(sequence)} – Arguments to pass to the event handler.
\end{itemize}

\textbf{Return type} bool or None

\textbf{Returns} (Since pyglet 1.2) \texttt{EVENT\_HANDLED} if an event handler returned \texttt{EVENT\_HANDLED}; \texttt{EVENT\_UNHANDLED} if one or more event handlers were invoked but returned only \texttt{EVENT\_UNHANDLED}; otherwise \texttt{False}. In pyglet 1.1 and earlier, the return value is always \texttt{None}.

\begin{verbatim}
FormattedDocument.event(*args)
Function decorator for an event handler.

Usage:

win = window.Window()

@win.event
def on_resize(self, width, height):
    # ...

or:

@win.event('on_resize')
def foo(self, width, height):
    # ...
\end{verbatim}

\begin{verbatim}
FormattedDocument.get_element(position)
Get the element at a specified position.

Parameters position (int) – Position in the document of the element.

Return type InlineElement
\end{verbatim}

\begin{verbatim}
FormattedDocument.get_paragraph_end(pos)
Get the end position of a paragraph.

Parameters pos (int) – Character position within paragraph.

Return type int
\end{verbatim}

\begin{verbatim}
FormattedDocument.get_paragraph_start(pos)
Get the starting position of a paragraph.

Parameters pos (int) – Character position within paragraph.

Return type int
\end{verbatim}

\begin{verbatim}
FormattedDocument.get_style_range(attribute, start, end)
Get an attribute style over the given range.

If the style varies over the range, \texttt{STYLE\_INDETERMINATE} is returned.

Parameters
\end{verbatim}
• `attribute` *(str)* – Name of style attribute to query.
• `start` *(int)* – Starting character position.
• `end` *(int)* – Ending character position (exclusive).

**Returns**  The style set for the attribute over the given range, or `STYLE_INDETERMINATE` if more than one value is set.

**FormattedDocument.insert_element** *(position, element, attributes=None)*
Insert a element into the document.

See the `InlineElement` class documentation for details of usage.

**Parameters**

• `position` *(int)* – Character insertion point within document.
• `element` *(InlineElement)* – Element to insert.
• `attributes` *(dict)* – Optional dictionary giving named style attributes of the inserted text.

**FormattedDocument.insert_text** *(start, text, attributes=None)*
Insert text into the document.

**Parameters**

• `start` *(int)* – Character insertion point within document.
• `text` *(str)* – Text to insert.
• `attributes` *(dict)* – Optional dictionary giving named style attributes of the inserted text.

**FormattedDocument.pop_handlers** ()
Pop the top level of event handlers off the stack.

**FormattedDocument.push_handlers** (*args, **kwargs)*
Push a level onto the top of the handler stack, then attach zero or more event handlers.

If keyword arguments are given, they name the event type to attach. Otherwise, a callable’s `_name_` attribute will be used. Any other object may also be specified, in which case it will be searched for callables with event names.

**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.

**FormattedDocument.remove_handler** *(name, handler)*
Remove a single event handler.

The given event handler is removed from the first handler stack frame it appears in. The handler must be the exact same callable as passed to `set_handler`, `set_handlers` or `push_handlers`; and the name must match the event type it is bound to.

No error is raised if the event handler is not set.

**Parameters**

• `name` *(str)* – Name of the event type to remove.
• `handler` *(callable)* – Event handler to remove.
FormattedDocument.**remove_handler**(*args, **kwargs*)
Remove event handlers from the event stack.

See **push_handler** for the accepted argument types. All handlers are removed from the first stack frame that contains any of the given handlers. No error is raised if any handler does not appear in that frame, or if no stack frame contains any of the given handlers.

If the stack frame is empty after removing the handlers, it is removed from the stack. Note that this interferes with the expected symmetry of **push_handler** and **pop_handler**.

FormattedDocument.**set_handler**(name, handler)
Attach a single event handler.

**Parameters**
- name (str) – Name of the event type to attach to.
- handler (callable) – Event handler to attach.

FormattedDocument.**set_handlers**(*args, **kwargs*)
Attach one or more event handlers to the top level of the handler stack.

See **push_handler** for the accepted argument types.

FormattedDocument.**set_paragraph_style**(start, end, attributes)
Set the style for a range of paragraphs.

This is a convenience method for **set_style** that aligns the character range to the enclosing paragraph(s).

**Parameters**
- start (int) – Starting character position.
- end (int) – Ending character position (exclusive).
- attributes (dict) – Dictionary giving named style attributes of the paragraphs.

FormattedDocument.**set_style**(start, end, attributes)
Set text style of some or all of the document.

**Parameters**
- start (int) – Starting character position.
- end (int) – Ending character position (exclusive).
- attributes (dict) – Dictionary giving named style attributes of the text.

**Events**

FormattedDocument.**on_delete_text**(start, end)
Text was deleted from the document.

**Parameters**
- start (int) – Starting character position of deleted text.
- end (int) – Ending character position of deleted text (exclusive).

FormattedDocument.**on_insert_text**(start, text)
Text was inserted into the document.

**Parameters**
• **start** *(int)* – Character insertion point within document.
• **text** *(str)* – The text that was inserted.

`FormattedDocument.on_style_text(start, end, attributes)`

Text character style was modified.

**Parameters**

• **start** *(int)* – Starting character position of modified text.
• **end** *(int)* – Ending character position of modified text (exclusive).
• **attributes** *(dict)* – Dictionary giving updated named style attributes of the text.

**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**

```python
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.

**Constructor:**

```python
__init__(ascent, descent, advance)
```

**Methods:**
place(layout, x, y)  Construct an instance of the element at the given coordinates.
remove(layout)  Remove this element from a layout.

Attributes:
position  Position of the element within the document.

Methods
InlineElement .place (layout, x, y)
Construct an instance of the element at the given coordinates.

Called when the element’s position within a layout changes, either due to the initial condition, changes in the
document or changes in the layout size.

It is the responsibility of the element to clip itself against the layout boundaries, and position itself appropriately
with respect to the layout’s position and viewport offset.

The TextLayout.top_state graphics state implements this transform and clipping into window space.

Parameters
• layout (pyglet.text.layout.TextLayout) – The layout the element moved within.
• x (int) – Position of the left edge of the element, relative to the left edge of the document, in
pixels.
• y (int) – Position of the baseline, relative to the top edge of the document, in pixels. Note
that this is typically negative.

InlineElement .remove (layout)
Remove this element from a layout.

The counterpart of place; called when the element is no longer visible in the given layout.

Parameters layout (pyglet.text.layout.TextLayout) – The layout the element was removed from.

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.
Constructor:
```python
__init__(text='')
```

Methods:
- `delete_text(start, end)` Delete text from the document.
- `get_element(position)` Get the element at a specified position.
- `get_element_runs()`
- `get_font([position, dpi])`
- `get_font_runs([dpi])`
- `get_paragraph_end(pos)` Get the end position of a paragraph.
- `get_paragraph_start(pos)` Get the starting position of a paragraph.
- `get_style(attribute[, position])`
- `get_style_range(attribute, start, end)` Get an attribute style over the given range.
- `get_style_runs(attribute)`
- `insert_element(position, element[, attributes])` Insert an element into the document.
- `insert_text(start, text[, attributes])` Insert text into the document.
- `set_paragraph_style(start, end, attributes)`
- `set_style(start, end, attributes)`

Events:
- `on_delete_text(start, end)` Text was deleted from the document.
- `on_insert_text(start, text)` Text was inserted into the document.
- `on_style_text(start, end, attributes)` Text character style was modified.

Attributes:
- `event_types` UnformattedDocument
- `text` Document text.

Methods
- `UnformattedDocument.get_element_runs()`
- `UnformattedDocument.get_font(position=None, dpi=None)`
- `UnformattedDocument.get_font_runs(dpi=None)`
- `UnformattedDocument.get_style(attribute, position=None)`
- `UnformattedDocument.get_style_runs(attribute)`
- `UnformattedDocument.set_paragraph_style(start, end, attributes)`
- `UnformattedDocument.set_style(start, end, attributes)`

Inherited members
Methods

UnformattedDocument.delete_text(start, end)
Delete text from the document.

Parameters
• start (int) – Starting character position to delete from.
• end (int) – Ending character position to delete to (exclusive).

UnformattedDocument.dispatch_event(event_type, *args)
Dispatch a single event to the attached handlers.

The event is propagated to all handlers from from the top of the stack until one returns EVENT_HANDLED. This method should be used only by EventDispatcher implementors; applications should call the dispatch_events method.

Since pyglet 1.2, the method returns EVENT_HANDLED if an event handler returned EVENT_HANDLED or EVENT_UNHANDLED if all events returned EVENT_UNHANDLED. If no matching event handlers are in the stack, False is returned.

Parameters
• event_type (str) – Name of the event.
• args (sequence) – Arguments to pass to the event handler.

Return type bool or None

Returns (Since pyglet 1.2) EVENT_HANDLED if an event handler returned EVENT_HANDLED; EVENT_UNHANDLED if one or more event handlers were invoked but returned only EVENT_UNHANDLED; otherwise False. In pyglet 1.1 and earlier, the return value is always None.

UnformattedDocument.event(*args)
Function decorator for an event handler.

Usage:
win = window.Window()

@win.event
def on_resize(self, width, height):
    # ...

or:

@win.event('on_resize')
def foo(self, width, height):
    # ...

UnformattedDocument.get_element(position)
Get the element at a specified position.

Parameters position (int) – Position in the document of the element.

Return type InlineElement

UnformattedDocument.get_paragraph_end(pos)
Get the end position of a paragraph.

Parameters pos (int) – Character position within paragraph.
Return type  int

UnformattedDocument.get_paragraph_start(pos)
Get the starting position of a paragraph.

Parameters  pos (int) – Character position within paragraph.

Return type  int

UnformattedDocument.get_style_range(attribute, start, end)
Get an attribute style over the given range.
If the style varies over the range, STYLE_INDETERMINATE is returned.

Parameters
• attribute (str) – Name of style attribute to query.
• start (int) – Starting character position.
• end (int) – Ending character position (exclusive).

Returns  The style set for the attribute over the given range, or STYLE_INDETERMINATE if more than one value is set.

UnformattedDocument.insert_element(position, element, attributes=None)
Insert a element into the document.
See the InlineElement class documentation for details of usage.

Parameters
• position (int) – Character insertion point within document.
• element (InlineElement) – Element to insert.
• attributes (dict) – Optional dictionary giving named style attributes of the inserted text.

UnformattedDocument.insert_text(start, text, attributes=None)
Insert text into the document.

Parameters
• start (int) – Character insertion point within document.
• text (str) – Text to insert.
• attributes (dict) – Optional dictionary giving named style attributes of the inserted text.

UnformattedDocument.pop_handlers()
Pop the top level of event handlers off the stack.

UnformattedDocument.push_handlers(*args, **kwargs)
Push a level onto the top of the handler stack, then attach zero or more event handlers.
If keyword arguments are given, they name the event type to attach. Otherwise, a callable’s
__name__ attribute will be used. Any other object may also be specified, in which case it will be searched for callables with event names.

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.

`UnformattedDocument.remove_handler(name, handler)`

Remove a single event handler.

The given event handler is removed from the first handler stack frame it appears in. The handler must be the exact same callable as passed to `set_handler`, `set_handlers` or `push_handlers`; and the name must match the event type it is bound to.

No error is raised if the event handler is not set.

Parameters

- name (str) – Name of the event type to remove.
- handler (callable) – Event handler to remove.

`UnformattedDocument.remove_handlers(*args, **kwargs)`

Remove event handlers from the event stack.

See `push_handlers` for the accepted argument types. All handlers are removed from the first stack frame that contains any of the given handlers. No error is raised if any handler does not appear in that frame, or if no stack frame contains any of the given handlers.

If the stack frame is empty after removing the handlers, it is removed from the stack. Note that this interferes with the expected symmetry of `push_handlers` and `pop_handlers`.

`UnformattedDocument.set_handler(name, handler)`

Attach a single event handler.

Parameters

- name (str) – Name of the event type to attach to.
- handler (callable) – Event handler to attach.

`UnformattedDocument.set_handlers(*args, **kwargs)`

Attach one or more event handlers to the top level of the handler stack.

See `push_handlers` for the accepted argument types.

Events

`UnformattedDocument.on_delete_text(start, end)`

Text was deleted from the document.

Parameters

- start (int) – Starting character position of deleted text.
- end (int) – Ending character position of deleted text (exclusive).

`UnformattedDocument.on_insert_text(start, text)`

Text was inserted into the document.

Parameters

- start (int) – Character insertion point within document.
- text (str) – The text that was inserted.

`UnformattedDocument.on_style_text(start, end, attributes)`

Text character style was modified.

Parameters
• **start** (*int*) – Starting character position of modified text.
• **end** (*int*) – Ending character position of modified text (exclusive).
• **attributes** (*dict*) – Dictionary giving updated named style attributes of the text.

**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**

- event
- re
- runlist
- sys

---

**pyglet.text.formats**  Document formats.

**Note:** Since pyglet 1.1

```
attributed  Extensible attributed text format for representing pyglet formatted documents.
html       Decode HTML into attributed text.
plaintext  Plain text decoder.
structured  Base class for structured (hierarchical) document formats.
```

**Modules**

**pyglet.text.formats.attributed**  Extensible attributed text format for representing pyglet formatted doc-

```
AttributedTextDecoder
```

**Classes**
AttributedTextDecoder Class

class AttributedTextDecoder

Methods:

- append(text)
- decode(text[, location])
- safe(ast)
- safe_node(node)

Methods

AttributedTextDecoder.append(text)
AttributedTextDecoder.decode(text, location=None)
AttributedTextDecoder.safe(ast)
AttributedTextDecoder.safe_node(node)

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

Decoder for HTML documents.

**Constructor:**

\[
__init__() \\
\]

Initialize and reset this instance.

**Methods:**

- `decode_structured(text, location)`
- `get_image(filename)`
- `handle_charref(name)`
- `handle_data(data)`
- `handle_endtag(tag)`
- `handle_entityref(name)`
- `handle_starttag(tag, case_attrs)`
- `prepare_for_data()`

**Attributes:**

- `CDATA_CONTENT_ELEMENTS`
- `default_style`
- `entitydefs`
- `Font_sizes`

**Methods**

HTMLDecoder.

- `decode_structured(text, location)`
- `get_image(filename)`
- `handle_charref(name)`
- `handle_data(data)`
- `handle_endtag(tag)`
- `handle_entityref(name)`
- `handle_starttag(tag, case_attrs)`
- `prepare_for_data()`

**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**

**Methods**

**HTMLDecoder.add_element**(*element*)

**HTMLDecoder.add_text**(*text*)

**HTMLDecoder.check_for_whole_start_tag**(*i*)

**HTMLDecoder.clear_cdata_mode**()

**HTMLDecoder.close**()

Handle any buffered data.

**HTMLDecoder.decode**(*text*, *location=None*)

**HTMLDecoder.error**(*message*)

**HTMLDecoder.feed**(*data*)

Feed data to the parser.

Call this as often as you want, with as little or as much text as you want (may include ‘n’).

**HTMLDecoder.get_starttag_text**()

Return full source of start tag: ‘<...>’.

**HTMLDecoder.getpos**()

Return current line number and offset.

**HTMLDecoder.goahead**(*end*)

**HTMLDecoder.handle_comment**(*data*)

**HTMLDecoder.handle_decl**(*decl*)

**HTMLDecoder.handle_pi**(*data*)

**HTMLDecoder.handle_startendtag**(*tag*, *attrs*)

**HTMLDecoder.parse_bogus_comment**(*i*, *report=1*)

**HTMLDecoder.parse_comment**(*i*, *report=1*)

**HTMLDecoder.parse_declaration**(*i*)

**HTMLDecoder.parse_endtag**(*i*)

**HTMLDecoder.parse_html_declaration**(*i*)

**HTMLDecoder.parse_marked_section**(*i*, *report=1*)

**HTMLDecoder.parse_pi**(*i*)

**HTMLDecoder.parse_starttag**(*i*)
HTMLDecoder.

```
HTMLDecoder.pop_style(key)
HTMLDecoder.push_style(key, styles)
HTMLDecoder.reset()
    Reset this instance. Loses all unprocessed data.
HTMLDecoder.set_cdata_mode(elem)
HTMLDecoder.unescape(s)
HTMLDecoder.unknown_decl(data)
HTMLDecoder.updatepos(i, j)
```

Attributes

```
HTMLDecoder.CDATA_CONTENT_ELEMENTS = ('script', 'style')
HTMLDecoder.entitydefs = None
```

Defined

- HTMLParser
- htmlentitydefs
- pyglet
- re
- structured

Notes

Plain text decoder.

```
pyglet.text.formats.plaintext
```

Classes

```
pyglet.text.DocumentDecoder  pyglet.text.formats.plaintext.PlainTextDecoder
```

PlainTextDecoder Class

class PlainTextDecoder

Methods:

```
decode(text[, location])
```

2.1. pyglet
Methods

`PlainTextDecoder.decode(text, location=None)`

Notes

- `pyglet`

### `pyglet.text.formats.structured`

Base class for structured (hierarchical) document formats.

```
pyglet.text.formats.structured
```

### Classes

```
pyglet.text.document.InlineElement
pyglet.text.formats.structured.ImageElement
```

#### `pyglet.text.formats.structured.ImageElement` Class

**class** `ImageElement` *(image, width=None, height=None)*

**Constructor:**

`__init__` *(image, width=None, height=None)*

**Methods:**

- `place(layout, x, y)`
- `remove(layout)`

**Attributes:**

- `position` Position of the element within the document.

#### Methods

- `ImageElement.place(layout, x, y)`
- `ImageElement.remove(layout)`

**Inherited members**
Attributes

ImageElement\texttt{.position}

Position of the element within the document. Read-only.

\hspace{1cm} Type \hspace{1cm} int

ListBuilder Class

class ListBuilder

Methods:

\begin{center}
\begin{tabular}{l|l}
\texttt{begin}(\texttt{decoder, style}) & Begin a list. \\
\texttt{get\_mark}([\texttt{value}]) & Get the mark text for the next list item. \\
\texttt{item}(\texttt{decoder, style[, value]}) & Begin a list item. \\
\end{tabular}
\end{center}

Methods

ListBuilder.\texttt{begin}(\texttt{decoder, style})

Begin a list.

Parameters

\begin{itemize}
\item \texttt{decoder} \hspace{2mm} (StructuredTextDecoder) \hspace{2mm} – Decoder.
\item \texttt{style} \hspace{2mm} (dict) \hspace{2mm} – Style dictionary that applies over the entire list.
\end{itemize}

ListBuilder.\texttt{get\_mark}(\texttt{value=}None)

Get the mark text for the next list item.

Parameters \hspace{1mm} \texttt{value} \hspace{2mm} (str) \hspace{2mm} – Optional value of the list item. The meaning is list-type dependent.

Return type \hspace{2mm} str

ListBuilder.\texttt{item}(\texttt{decoder, style, value=}None)

Begin a list item.

Parameters

\begin{itemize}
\item \texttt{decoder} \hspace{2mm} (StructuredTextDecoder) \hspace{2mm} – Decoder.
\item \texttt{style} \hspace{2mm} (dict) \hspace{2mm} – Style dictionary that applies over the list item.
\item \texttt{value} \hspace{2mm} (str) \hspace{2mm} – Optional value of the list item. The meaning is list-type dependent.
\end{itemize}
**OrderedListBuilder Class**

class `OrderedListBuilder` *(start, format)*

Constructor:

```python
__init__(start, format)
```

Create an ordered list with sequentially numbered mark text.

The format is composed of an optional prefix text, a numbering scheme character followed by suffix text. Valid numbering schemes are:

- Decimal Arabic
- Lowercase alphanumeric
- Uppercase alphanumeric
- Lowercase Roman
- Uppercase Roman

Prefix text may typically be ( or ) and suffix text is typically . or empty, but either can be any string.

**Parameters**

- `start` *(int)* – First list item number.
- `format` *(str)* – Format style, for example "1.".

**Methods**:

- `begin(decoder, style)` – Begin a list.
- `get_mark(value)`
- `item(decoder, style[, value])` – Begin a list item.

**Attributes**:

- `format_re`
Methods

OrderedListBuilder\texttt{.begin}(decoder, style)

Begin a list.

Parameters

\begin{itemize}
  \item \texttt{decoder} (StructuredTextDecoder) – Decoder.
  \item \texttt{style} (dict) – Style dictionary that applies over the entire list.
\end{itemize}

OrderedListBuilder\texttt{.item}(decoder, style, value=None)

Begin a list item.

Parameters

\begin{itemize}
  \item \texttt{decoder} (StructuredTextDecoder) – Decoder.
  \item \texttt{style} (dict) – Style dictionary that applies over the list item.
  \item \texttt{value} (str) – Optional value of the list item. The meaning is list-type dependent.
\end{itemize}

\begin{table}
\begin{tabular}{ll}
\texttt{pyglet.text.DocumentDecoder} & \texttt{pyglet.text.formats.structured.StructuredTextDecoder} \\
\end{tabular}
\end{table}

\begin{class}
\texttt{StructuredTextDecoder} Class
\end{class}

\begin{classclass}
\texttt{StructuredTextDecoder}
\end{classclass}

Methods:

\begin{itemize}
  \item \texttt{add_element}(element)
  \item \texttt{add_text}(text)
  \item \texttt{decode}(text[, location])
  \item \texttt{decode_structured}(text, location)
  \item \texttt{pop_style}(key)
  \item \texttt{push_style}(key, styles)
\end{itemize}

Methods

StructuredTextDecoder\texttt{.add_element}(element)
StructuredTextDecoder\texttt{.add_text}(text)
StructuredTextDecoder\texttt{.decode}(text, location=None)
StructuredTextDecoder\texttt{.decode_structured}(text, location)
StructuredTextDecoder\texttt{.pop_style}(key)
StructuredTextDecoder\texttt{.push_style}(key, styles)
UnorderedListBuilder Class

class UnorderedListBuilder(mark)

Constructor:

__init__(mark)
    Create an unordered list with constant mark text.

    Parameters mark (str) – Mark to prepend to each list item.

Methods:

begin(decoder, style)        # Begin a list.
get_mark(value)             # Begin a list item.

item(decoder, style[, value]) # Begin a list item.

Methods

UnorderedListBuilder.get_mark(value)

Inherited members

Methods

UnorderedListBuilder.begin(decoder, style)
    Begin a list.

    Parameters
    
    • decoder (StructuredTextDecoder) – Decoder.
    • style (dict) – Style dictionary that applies over the entire list.

UnorderedListBuilder.item(decoder, style, value=None)
    Begin a list item.

    Parameters
    
    • decoder (StructuredTextDecoder) – Decoder.
    • style (dict) – Style dictionary that applies over the list item.
    • value (str) – Optional value of the list item. The meaning is list-type dependent.
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

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IncrementalTextLayout</td>
<td>Displayed text suitable for interactive editing and/or scrolling large documents.</td>
</tr>
<tr>
<td>ScrollableTextLayout</td>
<td>Display text in a scrollable viewport.</td>
</tr>
<tr>
<td>ScrollableTextLayoutGroup</td>
<td>Top-level rendering group for ScrollableTextLayout.</td>
</tr>
<tr>
<td>TextLayout</td>
<td>Lay out and display documents.</td>
</tr>
<tr>
<td>TextLayoutForegroundDecorationGroup</td>
<td>Rendering group for decorative elements (e.g., glyph underlines) in all text layouts.</td>
</tr>
<tr>
<td>TextLayoutForegroundGroup</td>
<td>Rendering group for foreground elements (glyphs) in all text layouts.</td>
</tr>
<tr>
<td>TextLayoutGroup</td>
<td>Top-level rendering group for TextLayout.</td>
</tr>
<tr>
<td>TextLayoutTextureGroup</td>
<td>Rendering group for a glyph texture in all text layouts.</td>
</tr>
</tbody>
</table>

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.

**Constructor:**

```python
__init__(document, width, height, multiline=False, dpi=None, batch=None, group=None, wrap_lines=True)
```

**Methods:**

- `begin_update()` : Indicate that a number of changes to the layout or document are about to occur.
- `draw()` : Draw this text layout.
- `end_update()` : Perform pending layout changes since `begin_update`.
- `ensure_line_visible(line)` : Adjust `view_y` so that the line with the given index is visible.
- `ensure_x_visible(x)` : Adjust `view_x` so that the given X coordinate is visible.
- `get_line_count()` : Get the number of lines in the text layout.
- `get_line_from_point(x, y)` : Get the closest line index to a point.
- `get_line_from_position(position)` : Get the line index of a character position in the document.
- `get_point_from_line(line)` : Get the X, Y coordinates of a line index.
- `get_point_from_position(position[, line])` : Get the X, Y coordinates of a position in the document.
- `get_position_from_line(line)` : Get the closest document position of a given line index.
- `get_position_from_point(x, y)` : Get the closest document position to a point.
- `get_position_on_line(line, x)` : Get the closest document position for a given line index and X coordinate.
- `on_delete_text(start, end)` :  
- `on_insert_text(start, text)` :  
- `on_style_text(start, end, attributes)` :  
- `set_selection(start, end)` : Set the text selection range.

**Events:**

- `on_layout_update()` : Some or all of the layout text was reflowed.

**Attributes:**

- `anchor_x`
- `anchor_y`
- `content_valign` : Vertical alignment of content within larger layout box.
- `document`
- `dpi` : Get DPI used by this layout.
- `event_types`
- `foreground_decoration_group`
- `foreground_group`
- `height`
- `multiline`
- `selection_background_color` : Background color of active selection.
- `selection_color` : Text color of active selection.
- `selection_end` : End position of the active selection (exclusive).
- `selection_start` : Starting position of the active selection.

Continued on next page
Table 2.283 – continued from previous page

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>top_group</td>
<td></td>
</tr>
<tr>
<td>view_x</td>
<td>Horizontal scroll offset.</td>
</tr>
<tr>
<td>view_y</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
</tbody>
</table>

**Methods**

**IncrementalTextLayout.delete()**

Adjust `view_y` so that the line with the given index is visible.

**Parameters**

- `line (int)` – Line index.

**IncrementalTextLayout.ensure_line_visible(line)**

Adjust `view_y` so that the line with the given index is visible.

**Parameters**

- `line (int)` – Line index.

**IncrementalTextLayout.ensure_x_visible(x)**

Adjust `view_x` so that the given X coordinate is visible.

The X coordinate is given relative to the current `view_x`.

**Parameters**

- `x (int)` – X coordinate

**IncrementalTextLayout.getLine_count()**

Get the number of lines in the text layout.

**Return type**

int

**IncrementalTextLayout.getLine_from_point(x, y)**

Get the closest line index to a point.

**Parameters**

- `x (int)` – X coordinate.
- `y (int)` – Y coordinate.

**Return type**

int

**IncrementalTextLayout.getLine_from_position(position)**

Get the line index of a character position in the document.

**Parameters**

- `position (int)` – Document position.

**Return type**

int

**IncrementalTextLayout.getPoint_from_line(line)**

Get the X, Y coordinates of a line index.

**Parameters**

- `line (int)` – Line index.

**Return type**

(int, int)

**Returns** (x, y)

**IncrementalTextLayout.getPoint_from_position(position, line=None)**

Get the X, Y coordinates of a position in the document.

The position that ends a line has an ambiguous point: it can be either the end of the line, or the beginning of the next line. You may optionally specify a line index to disambiguate the case.

The resulting Y coordinate gives the baseline of the line.
Parameters

- **position** *(int)* – Character position within document.
- **line** *(int)* – Line index.

Return type *(int, int)*

Returns *(x, y)*

IncrementalTextLayout.get_position_from_line*(line)*

Get the first document character position of a given line index.

Parameters **line** *(int)* – Line index.

Return type **int**

IncrementalTextLayout.get_position_from_point*(x, y)*

Get the closest document position to a point.

Parameters

- **x** *(int)* – X coordinate
- **y** *(int)* – Y coordinate

IncrementalTextLayout.get_position_on_line*(line, x)*

Get the closest document position for a given line index and X coordinate.

Parameters

- **line** *(int)* – Line index.
- **x** *(int)* – X coordinate.

Return type **int**

IncrementalTextLayout.on_delete_text*(start, end)*

IncrementalTextLayout.on_insert_text*(start, text)*

IncrementalTextLayout.on_style_text*(start, end, attributes)*

IncrementalTextLayout.set_selection*(start, end)*

Set the text selection range.

If **start** equals **end** no selection will be visible.

Parameters

- **start** *(int)* – Starting character position of selection.
- **end** *(int)* – End of selection, exclusive.

Events

IncrementalTextLayout.on_layout_update()

Some or all of the layout text was reflowed.

Text reflow is caused by document edits or changes to the layout’s size. Changes to the layout’s position or active selection, and certain document edits such as text color, do not cause a reflow.

Handle this event to update the position of a graphical element that depends on the laid out position of a glyph or line.
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.begin_update()

- Indicate that a number of changes to the layout or document are about to occur.

Changes to the layout or document between calls to begin_update and end_update do not trigger any costly relayout of text. Relayout of all changes is performed when end_update is called.

Note that between the begin_update and end_update calls, values such as content_width and content_height are undefined (i.e., they may or may not be updated to reflect the latest changes).

IncrementalTextLayout.dispatch_event(event_type, *args)

- Dispatch a single event to the attached handlers.

The event is propagated to all handlers from from the top of the stack until one returns EVENT_HANDLED. This method should be used only by EventDispatcher implementors; applications should call the dispatch_events method.

Since pyglet 1.2, the method returns EVENT_HANDLED if an event handler returned EVENT_HANDLED or EVENT_UNHANDED if all events returned EVENT_UNHANDED. If no matching event handlers are in the stack, False is returned.

Parameters
- **event_type** (*str*) – Name of the event.
- **args** (*sequence*) – Arguments to pass to the event handler.

**Return type** bool or None

**Returns** (Since pyglet 1.2) `EVENT_HANDLED` if an event handler returned `EVENT_HANDLED`; `EVENT_UNHANDLED` if one or more event handlers were invoked but returned only `EVENT_UNHANDLED`; otherwise `False`. In pyglet 1.1 and earlier, the return value is always `None`.

IncrementalTextLayout.draw()

Draw this text layout.

Note that this method performs very badly if a batch was supplied to the constructor. If you add this layout to a batch, you should ideally use only the batch’s draw method.

IncrementalTextLayout.end_update()

Perform pending layout changes since `begin_update`.

See `begin_update`.

IncrementalTextLayout.event(*args)

Function decorator for an event handler.

**Usage:**

```python
win = window.Window()

@win.event
def on_resize(self, width, height):
    # ...
```

or:

```python
@win.event('on_resize')
def foo(self, width, height):
    # ...
```

IncrementalTextLayout.pop_handlers()

Pop the top level of event handlers off the stack.

IncrementalTextLayout.push_handlers(*args, **kwargs)

Push a level onto the top of the handler stack, then attach zero or more event handlers.

If keyword arguments are given, they name the event type to attach. Otherwise, a callable’s
`__name__` attribute will be used. Any other object may also be specified, in which case it will
be searched for callables with event names.

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.

IncrementalTextLayout.remove_handler(name, handler)

Remove a single event handler.

The given event handler is removed from the first handler stack frame it appears in. The handler
must be the exact same callable as passed to `set_handler`, `set_handlers` or `push_handlers`; and the
name must match the event type it is bound to.
No error is raised if the event handler is not set.

**Parameters**

- **name** (*str*) – Name of the event type to remove.
- **handler** (*callable*) – Event handler to remove.

**IncrementalTextLayout**.remove_handlers (*args, **kwargs*)

Remove event handlers from the event stack.

See `push_handlers` for the accepted argument types. All handlers are removed from the first stack frame that contains any of the given handlers. No error is raised if any handler does not appear in that frame, or if no stack frame contains any of the given handlers.

If the stack frame is empty after removing the handlers, it is removed from the stack. Note that this interferes with the expected symmetry of `push_handlers` and `pop_handlers`.

**IncrementalTextLayout**.set_handler (name, handler)

Attach a single event handler.

**Parameters**

- **name** (*str*) – Name of the event type to attach to.
- **handler** (*callable*) – Event handler to attach.

**IncrementalTextLayout**.set_handlers (*args, **kwargs*)

Attach one or more event handlers to the top level of the handler stack.

See `push_handlers` for the accepted argument types.

**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 \([0, \text{content_width} - \text{width}]\)

**Type** int

```
IncrementalTextLayout.x
IncrementalTextLayout.y
```

---

**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 \(x, y, \text{width}\) and \(\text{height}\); and offsets the text by a scroll offset.

Use \(\text{view}_x\) and \(\text{view}_y\) to scroll the text within the viewport.

**Constructor:**

```
__init__ (document, width, height, multiline=False, dpi=None, batch=None, group=None, wrap_lines=True)
```

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin_update()</td>
<td>Indicate that a number of changes to the layout or document are about to occur.</td>
</tr>
<tr>
<td>delete()</td>
<td>Remove this layout from its batch.</td>
</tr>
<tr>
<td>draw()</td>
<td>Draw this text layout.</td>
</tr>
<tr>
<td>end_update()</td>
<td>Perform pending layout changes since \text{begin_update}.</td>
</tr>
<tr>
<td>on_delete_text(start, end)</td>
<td>Event handler for \text{AbstractDocument.on_delete_text}.</td>
</tr>
<tr>
<td>on_insert_text(start, text)</td>
<td>Event handler for \text{AbstractDocument.on_insert_text}.</td>
</tr>
<tr>
<td>on_style_text(start, end, attributes)</td>
<td>Event handler for \text{AbstractDocument.on_style_text}.</td>
</tr>
</tbody>
</table>

**Attributes:**

```
anchor_x
```

Continued on next page
Table 2.285 – continued from previous page

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>anchor_y</code></td>
<td>Vertical alignment of content within larger layout box.</td>
</tr>
<tr>
<td><code>content_valign</code></td>
<td>Vertical alignment of content within larger layout box.</td>
</tr>
<tr>
<td><code>document</code></td>
<td>Get DPI used by this layout.</td>
</tr>
<tr>
<td><code>dpi</code></td>
<td>Get DPI used by this layout.</td>
</tr>
<tr>
<td><code>foreground Decoration Group</code></td>
<td>Foreground decoration group.</td>
</tr>
<tr>
<td><code>foreground Group</code></td>
<td>Foreground group.</td>
</tr>
<tr>
<td><code>height</code></td>
<td>Set if multiline layout is enabled.</td>
</tr>
<tr>
<td><code>multiline</code></td>
<td>Set if multiline layout is enabled.</td>
</tr>
<tr>
<td><code>top Group</code></td>
<td>Top group.</td>
</tr>
<tr>
<td><code>view_x</code></td>
<td>Horizontal scroll offset.</td>
</tr>
<tr>
<td><code>view_y</code></td>
<td>Vertical scroll offset.</td>
</tr>
<tr>
<td><code>width</code></td>
<td>Width.</td>
</tr>
<tr>
<td><code>x</code></td>
<td>X.</td>
</tr>
<tr>
<td><code>y</code></td>
<td>Y.</td>
</tr>
</tbody>
</table>

### 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 $[0, \text{content_width} - \text{width}]$.
  - **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 $[\text{height} - \text{content_height}, 0]$ are automatically clipped in range.
  - **Type**: int
- **ScrollableTextLayout**.width
- **ScrollableTextLayout**.x
- **ScrollableTextLayout**.y

### Inherited members

### Methods

- **ScrollableTextLayout**.begin_update()
  - Indicate that a number of changes to the layout or document are about to occur.
Changes to the layout or document between calls to `begin_update` and `end_update` do not trigger any costly relayout of text. Relayout of all changes is performed when `end_update` is called.

Note that between the `begin_update` and `end_update` calls, values such as `content_width` and `content_height` are undefined (i.e., they may or may not be updated to reflect the latest changes).

```python
ScrollableTextLayout.delete()
Remove this layout from its batch.
```

```python
ScrollableTextLayout.draw()
Draw this text layout.
Note that this method performs very badly if a batch was supplied to the constructor. If you add this layout to a batch, you should ideally use only the batch’s draw method.
```

```python
ScrollableTextLayout.end_update()
Perform pending layout changes since `begin_update`.
See `begin_update`.
```

```python
ScrollableTextLayout.on_delete_text(start, end)
Event handler for `AbstractDocument.on_delete_text`.
The event handler is bound by the text layout; there is no need for applications to interact with this method.
```

```python
ScrollableTextLayout.on_insert_text(start, text)
Event handler for `AbstractDocument.on_insert_text`.
The event handler is bound by the text layout; there is no need for applications to interact with this method.
```

```python
ScrollableTextLayout.on_style_text(start, end, attributes)
Event handler for `AbstractDocument.on_style_text`.
The event handler is bound by the text layout; there is no need for applications to interact with this method.
```

### Attributes

```python
ScrollableTextLayout.background_group = OrderedGroup(0)
```

```python
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
```

```python
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>

---

**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.

**Constructor:**

__init__ (parent=None)
Create a group.

Parameters parent (Group) – Group to contain this group; its state will be set before this state’s.

**Methods:**

- set_state()
- unset_state()

**Attributes:**

- height
  Height of the text layout.
- left
  Left edge of the text layout.
- top
  Top edge of the text layout (measured from the bottom of the graphics viewport).
- translate_x
- translate_y

Continued on next page
Table 2.287 – continued from previous page

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>view_x</td>
<td>Horizontal scroll offset.</td>
</tr>
<tr>
<td>view_y</td>
<td>Vertical scroll offset.</td>
</tr>
<tr>
<td>width</td>
<td>Width of the text layout.</td>
</tr>
</tbody>
</table>

Methods

ScrollViewTextLayoutGroup.set_state()
ScrollViewTextLayoutGroup.unset_state()

Attributes

ScrollViewTextLayoutGroup.height
  Height of the text layout.
  Type int
ScrollViewTextLayoutGroup.left
  Left edge of the text layout.
  Type int
ScrollViewTextLayoutGroup.top
  Top edge of the text layout (measured from the bottom of the graphics viewport).
  Type int
ScrollViewTextLayoutGroup.translate_x = 0
ScrollViewTextLayoutGroup.translate_y = 0
ScrollViewTextLayoutGroup.view_x
  Horizontal scroll offset.
  Type int
ScrollViewTextLayoutGroup.view_y
  Vertical scroll offset.
  Type int
ScrollViewTextLayoutGroup.width
  Width of the text layout.
  Type int

Inherited members

Methods

ScrollViewTextLayoutGroup.set_state_recursive()
  Set this group and its ancestry.
  Call this method if you are using a group in isolation: the parent groups will be called in top-down order, with this class’s set being called last.
ScrollViewLayoutGroup.unset_state_recursive()
Unset this group and its ancestry.
The inverse of set_state_recursive.

TextLayout Class

```python
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.

Constructor:

```python
__init__ (document, width=None, height=None, multiline=False, dpi=None, batch=None, group=None, wrap_lines=True)
```
Create a text layout.

Parameters

- `width` (int) – Width of the layout in pixels, or None
- `height` (int) – Height of the layout in pixels, or None
- `multiline` (bool) – If False, newline and paragraph characters are ignored, and text is not word-wrapped. If True, text is wrapped only if the `wrap_lines` is True.
- `dpi` (float) – Font resolution; defaults to 96.
- `batch` (Batch) – Optional graphics batch to add this layout to.
• **group** *(Group)* – Optional rendering group to parent all groups this text layout uses. Note that layouts with different rendered simultaneously in a batch.

• **wrap_lines** *(bool)* – If True and *multiline* is True, the text is word-wrapped using the specified width.

### Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>begin_update()</code></td>
<td>Indicate that a number of changes to the layout or document are about to occur.</td>
</tr>
<tr>
<td><code>delete()</code></td>
<td>Remove this layout from its batch.</td>
</tr>
<tr>
<td><code>draw()</code></td>
<td>Draw this text layout.</td>
</tr>
<tr>
<td><code>end_update()</code></td>
<td>Perform pending layout changes since <code>begin_update</code>.</td>
</tr>
<tr>
<td><code>on_delete_text(start, end)</code></td>
<td>Event handler for <code>AbstractDocument.on_delete_text</code>.</td>
</tr>
<tr>
<td><code>on_insert_text(start, text)</code></td>
<td>Event handler for <code>AbstractDocument.on_insert_text</code>.</td>
</tr>
<tr>
<td><code>on_style_text(start, end, attributes)</code></td>
<td>Event handler for <code>AbstractDocument.on_style_text</code>.</td>
</tr>
</tbody>
</table>

### Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>anchor_x</code></td>
<td>Horizontal anchor alignment.</td>
</tr>
<tr>
<td><code>anchor_y</code></td>
<td>Vertical anchor alignment.</td>
</tr>
<tr>
<td><code>content_valign</code></td>
<td>Vertical alignment of content within larger layout box.</td>
</tr>
<tr>
<td><code>document</code></td>
<td>Get DPI used by this layout.</td>
</tr>
<tr>
<td><code>foreground_decoration_group</code></td>
<td></td>
</tr>
<tr>
<td><code>foreground_group</code></td>
<td></td>
</tr>
<tr>
<td><code>height</code></td>
<td>Height of the layout.</td>
</tr>
<tr>
<td><code>multiline</code></td>
<td>Set if multiline layout is enabled.</td>
</tr>
<tr>
<td><code>top_group</code></td>
<td></td>
</tr>
<tr>
<td><code>width</code></td>
<td>Width of the layout.</td>
</tr>
<tr>
<td><code>x</code></td>
<td>X coordinate of the layout.</td>
</tr>
<tr>
<td><code>y</code></td>
<td>Y coordinate of the layout.</td>
</tr>
</tbody>
</table>

### Methods

*`TextLayout.begin_update()`*

Indicate that a number of changes to the layout or document are about to occur.

Changes to the layout or document between calls to `begin_update` and `end_update` do not trigger any costly relayout of text. Relayout of all changes is performed when `end_update` is called.

Note that between the `begin_update` and `end_update` calls, values such as `content_width` and `content_height` are undefined (i.e., they may or may not be updated to reflect the latest changes).

*`TextLayout.delete()`*

Remove this layout from its batch.

*`TextLayout.draw()`*

Draw this text layout.

Note that this method performs very badly if a batch was supplied to the constructor. If you add this layout to a batch, you should ideally use only the batch’s `draw` method.

*`TextLayout.end_update()`*

Perform pending layout changes since `begin_update`. 
See `begin_update`.

**TextLayout.on_delete_text** (*start, end*)

Event handler for `AbstractDocument.on_delete_text`.

The event handler is bound by the text layout; there is no need for applications to interact with this method.

**TextLayout.on_insert_text** (*start, text*)

Event handler for `AbstractDocument.on_insert_text`.

The event handler is bound by the text layout; there is no need for applications to interact with this method.

**TextLayout.on_style_text** (*start, end, attributes*)

Event handler for `AbstractDocument.on_style_text`.

The event handler is bound by the text layout; there is no need for applications to interact with this method.

**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.
Constructor:

```python
__init__(order, parent=None)
Create an ordered group.
```

Parameters

- **order** (*int*) – Order of this group.
- **parent** (*Group*) – Parent of this group.

Methods:

```python
set_state()
```

Methods

`TextLayoutForegroundDecorationGroup.set_state()`

Inherited members

Methods

`TextLayoutForegroundDecorationGroup.set_state_recursive()`  
Set this group and its ancestry.

Call this method if you are using a group in isolation: the parent groups will be called in top-down order, with this class’s `set` being called last.

`TextLayoutForegroundDecorationGroup.unset_state()`  
Repeal the OpenGL state change.

The default implementation does nothing.

`TextLayoutForegroundDecorationGroup.unset_state_recursive()`  
Unset this group and its ancestry.

The inverse of `set_state_recursive`.

---

`TextLayoutForegroundGroup Class`

class `TextLayoutForegroundGroup` *(order, parent=None)*

Rendering group for foreground elements (glyphs) in all text layouts.

The group enables `GL_TEXTURE_2D`.

Constructor:

```python
__init__(order, parent=None)
Create an ordered group.
```

Parameters

---
• **order** *(int)* – Order of this group.
• **parent** *(Group)* – Parent of this group.

**Methods**

```python
set_state()
```

**Methods**

**TextLayoutForegroundGroup**. **set_state**()

**Inherited members**

**Methods**

**TextLayoutForegroundGroup**. **set_state_recursive**()

Set this group and its ancestry.

Call this method if you are using a group in isolation: the parent groups will be called in top-down order, with this class's `set` being called last.

**TextLayoutForegroundGroup**. **unset_state**()

Repeal the OpenGL state change.

The default implementation does nothing.

**TextLayoutForegroundGroup**. **unset_state_recursive**()

Unset this group and its ancestry.

The inverse of `set_state_recursive`.

---

**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.

**Constructor:**

```python
__init__(parent=None)
```

Create a group.

**Parameters** **parent** *(Group)* – Group to contain this group; its state will be set before this state's.
Methods:

- `set_state()`
- `unset_state()`

Methods

TextLayoutGroup.

- `set_state()`
- `unset_state()`

Inherited members

Methods

TextLayoutGroup.

- `set_state_recursive()`
  Set this group and its ancestry.
  Call this method if you are using a group in isolation: the parent groups will be called in top-down order, with this class’s `set` being called last.

- `unset_state_recursive()`
  Unset this group and its ancestry.
  The inverse of `set_state_recursive`.

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.

Constructor:

- `__init__` (texture, parent)

Methods:

- `set_state()`

Methods
TextLayoutTextureGroup.set_state()

Inherited members

Methods

TextLayoutTextureGroup.set_state_recursive()
Set this group and its ancestry.

Call this method if you are using a group in isolation: the parent groups will be called in top-down
order, with this class’s set being called last.

TextLayoutTextureGroup.unset_state()
Repeal the OpenGL state change.

The default implementation does nothing.

TextLayoutTextureGroup.unset_state_recursive()
Unset this group and its ancestry.

The inverse of set_state_recursive.

Variables
compat_platform = ‘linux2’
str(object='') -> string

Return a nice string representation of the object. If the argument is a string, the return value is the same object.

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
Table 2.294 – continued from previous page

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RunIterator</td>
<td>List of contiguous runs of values.</td>
</tr>
<tr>
<td>RunList</td>
<td>List of contiguous runs of values.</td>
</tr>
<tr>
<td>ZipRunIterator</td>
<td>Iterate over multiple run iterators concurrently.</td>
</tr>
</tbody>
</table>

Classes

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:

```python
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:

```python
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.

Methods:

```
ranges(start, end)  # Iterate over a subrange of the run list.
```

Methods

AbstractRunIterator.ranges(start, end)

Iterate over a subrange of the run list.

See the class documentation for examples of valid usage.
Parameters

- **start** *(int)* – Start index to iterate from.
- **end** *(int)* – End index, exclusive.

Return type: iterator

Returns: Iterator over (start, end, value) tuples.

---

**ConstRunIterator Class**

```python
class ConstRunIterator(length, value)
```

Iterate over a constant value without creating a RunList.

**Constructor:**

```python
__init__(length, value)
```

**Methods:**

```python
next()
```

```python
ranges(start, end)
```

---

**Methods**

- `ConstRunIterator.next()`
- `ConstRunIterator.ranges(start, end)`

---

**FilteredRunIterator Class**

```python
class FilteredRunIterator(base_iterator, filter, default)
```

Iterate over an `AbstractRunIterator` with filtered values replaced by a default value.

**Constructor:**

```python
__init__(base_iterator, filter, default)
```

Create a filtered run iterator.

**Parameters**

- **base_iterator** *(AbstractRunIterator)* – Source of runs.
• **filter** (*lambda object*) – Function taking a value as parameter, and returning `True` if the value is acceptable, and `False` if the default value should be substituted.

• **default** (*object*) – Default value to replace filtered values.

Methods:

```
ranges(start, end)
```

Methods

`FilteredRunIterator`.

```
ranges (start, end)
```

```
pyglet.text.runlist.AbstractRunIterator ➔ pyglet.text.runlist.OverrideRunIterator
```

**OverrideRunIterator** Class

class `OverrideRunIterator` (*base_iterator, start, end, value*)

Iterator over a `RunIterator`, with a value temporarily replacing a given range.

Constructor:

```
__init__ (base_iterator, start, end, value)
```

Create a derived iterator.

Parameters

• **start** (*int*) – Start of range to override

• **end** (*int*) – End of range to override, exclusive

• **value** (*object*) – Value to replace over the range

Methods:

```
ranges(start, end)
```

Methods

`OverrideRunIterator`.

```
ranges (start, end)
```
RunIterator Class

class RunIterator (run_list)

Constructor:
__init__ (run_list)

Methods:

next()

ranges(start, end)

Methods
RunIterator.next()
RunIterator.ranges(start, end)

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 "aaaabbcccccc" 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 "aaxxbcccccc."

Constructor:
__init__ (size, initial)

Create a run list of the given size and a default value.

Parameters

- **size** (int) – Number of characters to represent initially.
• **initial** *(object)* – The value of all characters in the run list.

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>delete(start, end)</code></td>
<td>Remove characters from the run list.</td>
</tr>
<tr>
<td><code>get_run_iterator()</code></td>
<td>Get an extended iterator over the run list.</td>
</tr>
<tr>
<td><code>insert(pos, length)</code></td>
<td>Insert characters into the run list.</td>
</tr>
<tr>
<td><code>set_run(start, end, value)</code></td>
<td>Set the value of a range of characters.</td>
</tr>
</tbody>
</table>

**Methods**

`RunList.delete(start, end)`

Remove characters from the run list.

**Parameters**

- `start (int)` – Starting index to remove from.
- `end (int)` – End index, exclusive.

`RunList.get_run_iterator()`

Get an extended iterator over the run list.

**Return type** `RunIterator`

`RunList.insert(pos, length)`

Insert characters into the run list.

The inserted characters will take on the value immediately preceding the insertion point (or the value of the first character, if `pos` is 0).

**Parameters**

- `pos (int)` – Insertion index
- `length (int)` – Number of characters to insert.

`RunList.set_run(start, end, value)`

Set the value of a range of characters.

**Parameters**

- `start (int)` – Start index of range.
- `end (int)` – End of range, exclusive.
- `value (object)` – Value to set over the range.

pyglet.text.runlist.AbstractRunIterator  ➔ pyglet.text.runlist.ZipRunIterator

**ZipRunIterator Class**
class **ZipRunIterator** (*range_iterators*)

Iterate over multiple run iterators concurrently.

**Constructor:**

```python
__init__(range_iterators)
```

**Methods:**

```python
ranges(start, end)
```

**Methods**

**ZipRunIterator**. **ranges** (*start, end*)

**Classes**

```python
<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocumentLabel</td>
<td>Base label class.</td>
</tr>
<tr>
<td>HTMLLabel</td>
<td>HTML formatted text label.</td>
</tr>
<tr>
<td>Label</td>
<td>Plain text label.</td>
</tr>
</tbody>
</table>
```

**DocumentDecoder Class**

```python
class DocumentDecoder
    Abstract document decoder.

Methods:

```python
decode(text[, location])  Decode document text.
```

**Methods**

**DocumentDecoder**. **decode** (*text*, *location=None*)

Decode document text.

**Parameters**

- **text** (*str*) – Text to decode
- **location** (*Location*) – Location to use as base path for additional resources referenced within the document (for example, HTML images).

**Return type** *AbstractDocument*
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.

Constructor:

__init__ (document=None, x=0, y=0, width=None, height=None, anchor_x='left', anchor_y='baseline', multiline=False, dpi=None, batch=None, group=None)

Create a label for a given document.

Parameters

• document (AbstractDocument) – Document to attach to the layout.
• x (int) – X coordinate of the label.
• y (int) – Y coordinate of the label.
• width (int) – Width of the label in pixels, or None
• height (int) – Height of the label in pixels, or None
• anchor_x (str) – Anchor point of the X coordinate: one of "left", "center" or "right".
• anchor_y (str) – Anchor point of the Y coordinate: one of "bottom", "baseline", "center" or "top".
• multiline (bool) – If True, the label will be word-wrapped and accept newline characters. You must also set the width of the label.
• dpi (float) – Resolution of the fonts in this layout. Defaults to 96.
• batch (Batch) – Optional graphics batch to add the label to.
• group (Group) – Optional graphics group to use.

Methods:

get_style(name) Get a document style value by name.
set_style(name, value) Set a document style value by name over the whole document.

Attributes:

anchor_x Horizontal anchor alignment.
anchor_y Vertical anchor alignment.
bold Bold font style.
Table 2.305 – continued from previous page

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
<td>Text color.</td>
</tr>
<tr>
<td>content_valign</td>
<td>Vertical alignment of content within larger layout box.</td>
</tr>
<tr>
<td>document</td>
<td></td>
</tr>
<tr>
<td>dpi</td>
<td>Get DPI used by this layout.</td>
</tr>
<tr>
<td>font_name</td>
<td>Font family name.</td>
</tr>
<tr>
<td>font_size</td>
<td>Font size, in points.</td>
</tr>
<tr>
<td>height</td>
<td>Height of the layout.</td>
</tr>
<tr>
<td>italic</td>
<td>Italic font style.</td>
</tr>
<tr>
<td>multiline</td>
<td>Set if multiline layout is enabled.</td>
</tr>
<tr>
<td>text</td>
<td>The text of the label.</td>
</tr>
<tr>
<td>width</td>
<td>Width of the layout.</td>
</tr>
<tr>
<td>x</td>
<td>X coordinate of the layout.</td>
</tr>
<tr>
<td>y</td>
<td>Y coordinate of the layout.</td>
</tr>
</tbody>
</table>

**Methods**

DocumentLabel. `get_style(name)`

Get a document style value by name.

If the document has more than one value of the named style, `pyglet.text.document.STYLE_INDETERMINATE` is returned.

**Parameters**

- `name` (**str**) – Style name to query. See documentation for `pyglet.text.layout` for known style names.

**Return type** **object**

DocumentLabel. `set_style(name, value)`

Set a document style value by name over the whole document.

**Parameters**

- `name` (**str**) – Name of the style to set. See documentation for `pyglet.text.layout` for known style names.
- `value` (**object**) – Value of the style.

**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**

2.1. **pyglet**
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

Methods

DocumentLabel.begin_update()
   Indicate that a number of changes to the layout or document are about to occur.

   Changes to the layout or document between calls to begin_update and end_update do not trigger
   any costly relayout of text. Relayout of all changes is performed when end_update is called.

   Note that between the begin_update and end_update calls, values such as content_width and content_height
   are undefined (i.e., they may or may not be updated to reflect the latest changes).

DocumentLabel.delete()
   Remove this layout from its batch.

DocumentLabel.draw()
   Draw this text layout.

   Note that this method performs very badly if a batch was supplied to the constructor. If you add this
   layout to a batch, you should ideally use only the batch’s draw method.

DocumentLabel.end_update()
   Perform pending layout changes since begin_update.

   See begin_update.

DocumentLabel.on_delete_text(start, end)
   Event handler for AbstractDocument.on_delete_text.

   The event handler is bound by the text layout; there is no need for applications to interact with this
   method.

DocumentLabel.on_insert_text(start, text)
   Event handler for AbstractDocument.on_insert_text.

   The event handler is bound by the text layout; there is no need for applications to interact with this
   method.

DocumentLabel.on_style_text(start, end, attributes)
   Event handler for AbstractDocument.on_style_text.

   The event handler is bound by the text layout; there is no need for applications to interact with this
   method.
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 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

```
pyglet.text.layout.TextLayout ─> pyglet.text.DocumentLabel ─> pyglet.text.HTMLLabel
```

### 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.

**Constructor:**

```
__init__ (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)
```

Create a label with an HTML string.

**Parameters**

- *text (str)* – HTML formatted text to display.
• **location** (*Location*) – Location object for loading images referred to in the document. By default, the working directory is used.

• **x** (*int*) – X coordinate of the label.

• **y** (*int*) – Y coordinate of the label.

• **width** (*int*) – Width of the label in pixels, or None

• **height** (*int*) – Height of the label in pixels, or None

• **anchor_x** (*str*) – Anchor point of the X coordinate: one of "left", "center" or "right".

• **anchor_y** (*str*) – Anchor point of the Y coordinate: one of "bottom", "baseline", "center" or "top".

• **multiline** (*bool*) – If True, the label will be word-wrapped and render paragraph and line breaks. You must also set the width of the label.

• **dpi** (*float*) – Resolution of the fonts in this layout. Defaults to 96.

• **batch** (*Batch*) – Optional graphics batch to add the label to.

• **group** (*Group*) – Optional graphics group to use.

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_style(name)</code></td>
<td>Get a document style value by name.</td>
</tr>
<tr>
<td><code>set_style(name, value)</code></td>
<td>Set a document style value by name over the whole document.</td>
</tr>
</tbody>
</table>

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor_x</td>
<td>Horizontal anchor alignment.</td>
</tr>
<tr>
<td>anchor_y</td>
<td>Vertical anchor alignment.</td>
</tr>
<tr>
<td>bold</td>
<td>Bold font style.</td>
</tr>
<tr>
<td>color</td>
<td>Text color.</td>
</tr>
<tr>
<td>content_valign</td>
<td>Vertical alignment of content within larger layout box.</td>
</tr>
<tr>
<td>document</td>
<td></td>
</tr>
<tr>
<td>dpi</td>
<td>Get DPI used by this layout.</td>
</tr>
<tr>
<td>font_name</td>
<td>Font family name.</td>
</tr>
<tr>
<td>font_size</td>
<td>Font size, in points.</td>
</tr>
<tr>
<td>height</td>
<td>Height of the layout.</td>
</tr>
<tr>
<td>italic</td>
<td>Italic font style.</td>
</tr>
<tr>
<td>multiline</td>
<td>Set if multiline layout is enabled.</td>
</tr>
<tr>
<td>text</td>
<td>HTML formatted text of the label.</td>
</tr>
<tr>
<td>width</td>
<td>Width of the layout.</td>
</tr>
<tr>
<td>x</td>
<td>X coordinate of the layout.</td>
</tr>
<tr>
<td>y</td>
<td>Y coordinate of the layout.</td>
</tr>
</tbody>
</table>

**Attributes**

`HTMLLabel.text`

HTML formatted text of the label.
Inherited members

Methods

HTMLLabel.begin_update()
Indicate that a number of changes to the layout or document are about to occur.

Changes to the layout or document between calls to begin_update and end_update do not trigger any costly relayout of text. Relayout of all changes is performed when end_update is called.

Note that between the begin_update and end_update calls, values such as content_width and content_height are undefined (i.e., they may or may not be updated to reflect the latest changes).

HTMLLabel.delete()
Remove this layout from its batch.

HTMLLabel.draw()
Draw this text layout.

Note that this method performs very badly if a batch was supplied to the constructor. If you add this layout to a batch, you should ideally use only the batch’s draw method.

HTMLLabel.end_update()
Perform pending layout changes since begin_update.

See begin_update.

HTMLLabel.get_style(name)
Get a document style value by name.

If the document has more than one value of the named style, pyglet.text.document.STYLE_INDETERMINATE is returned.

Parameters name (str) – Style name to query. See documentation for pyglet.text.layout for known style names.

Return type object

HTMLLabel.on_delete_text(start, end)
Event handler for AbstractDocument.on_delete_text.

The event handler is bound by the text layout; there is no need for applications to interact with this method.

HTMLLabel.on_insert_text(start, text)
Event handler for AbstractDocument.on_insert_text.

The event handler is bound by the text layout; there is no need for applications to interact with this method.

HTMLLabel.on_style_text(start, end, attributes)
Event handler for AbstractDocument.on_style_text.

The event handler is bound by the text layout; there is no need for applications to interact with this method.

HTMLLabel.set_style(name, value)
Set a document style value by name over the whole document.

Parameters
• **name** (*str*) – Name of the style to set. See documentation for *pyglet.text.layout* for known style names.
• **value** (*object*) – Value of the style.

**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:

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.

Constructor:

__init__:

( 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 )

Create a plain text label.

Parameters

• text (str) – Text to display.
• font_name (str or list) – Font family name(s). If more than one name is given, the first matching name is used.
• font_size (float) – Font size, in points.
• bold (bool) – Bold font style.
• italic (bool) – Italic font style.
• color ((int, int, int, int)) – Font colour, as RGBA components in range [0, 255].
• x (int) – X coordinate of the label.
• y (int) – Y coordinate of the label.
• width (int) – Width of the label in pixels, or None
• height (int) – Height of the label in pixels, or None
• anchor_x (str) – Anchor point of the X coordinate: one of "left", "center" or "right".
• anchor_y (str) – Anchor point of the Y coordinate: one of "bottom", "baseline", "center" or "top".
• align (str) – Horizontal alignment of text on a line, only applies if a width is supplied. One of "left", "center" or "right".
• multiline (bool) – If True, the label will be word-wrapped and accept newline characters. You must also set the width of the label.
• **dpi (float)** – Resolution of the fonts in this layout. Defaults to 96.

• **batch (Batch)** – Optional graphics batch to add the label to.

• **group (Group)** – Optional graphics group to use.

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_style(name)</td>
<td>Get a document style value by name.</td>
</tr>
<tr>
<td>set_style(name, value)</td>
<td>Set a document style value by name over the whole document.</td>
</tr>
</tbody>
</table>

**Attributes:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anchor_x</td>
<td>Horizontal anchor alignment.</td>
</tr>
<tr>
<td>anchor_y</td>
<td>Vertical anchor alignment.</td>
</tr>
<tr>
<td>bold</td>
<td>Bold font style.</td>
</tr>
<tr>
<td>color</td>
<td>Text color.</td>
</tr>
<tr>
<td>content_valign</td>
<td>Vertical alignment of content within larger layout box.</td>
</tr>
<tr>
<td>document</td>
<td>Get DPI used by this layout.</td>
</tr>
<tr>
<td>font_name</td>
<td>Font family name.</td>
</tr>
<tr>
<td>font_size</td>
<td>Font size, in points.</td>
</tr>
<tr>
<td>height</td>
<td>Height of the layout.</td>
</tr>
<tr>
<td>italic</td>
<td>Italic font style.</td>
</tr>
<tr>
<td>multiline</td>
<td>Set if multiline layout is enabled.</td>
</tr>
<tr>
<td>text</td>
<td>The text of the label.</td>
</tr>
<tr>
<td>width</td>
<td>Width of the layout.</td>
</tr>
<tr>
<td>x</td>
<td>X coordinate of the layout.</td>
</tr>
<tr>
<td>y</td>
<td>Y coordinate of the layout.</td>
</tr>
</tbody>
</table>

**Inherited members**

**Methods**

**Label.begin_update()**

Indicate that a number of changes to the layout or document are about to occur.

Changes to the layout or document between calls to `begin_update` and `end_update` do not trigger any costly relayout of text. Relayout of all changes is performed when `end_update` is called.

Note that between the `begin_update` and `end_update` calls, values such as `content_width` and `content_height` are undefined (i.e., they may or may not be updated to reflect the latest changes).

**Label.delete()**

Remove this layout from its batch.

**Label.draw()**

Draw this text layout.

Note that this method performs very badly if a batch was supplied to the constructor. If you add this layout to a batch, you should ideally use only the batch’s draw method.
Label.end_update()
    Perform pending layout changes since begin_update.
    
    See begin_update.
Label.get_style(name)
    Get a document style value by name.
    
    If the document has more than one value of the named style, pyglet.text.document.STYLE_INDETERMINATE is returned.
    
    Parameters name (str) – Style name to query. See documentation for pyglet.text.layout for known style names.
    
    Return type object
Label.on_delete_text(start, end)
    Event handler for AbstractDocument.on_delete_text.
    
    The event handler is bound by the text layout; there is no need for applications to interact with this method.
Label.on_insert_text(start, text)
    Event handler for AbstractDocument.on_insert_text.
    
    The event handler is bound by the text layout; there is no need for applications to interact with this method.
Label.on_style_text(start, end, attributes)
    Event handler for AbstractDocument.on_style_text.
    
    The event handler is bound by the text layout; there is no need for applications to interact with this method.
Label.set_style(name, value)
    Set a document style value by name over the whole document.
    
    Parameters
        • name (str) – Name of the style to set. See documentation for pyglet.text.layout for known style names.
        • value (object) – Value of the style.

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

2.1. pyglet
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

Labelmultiline
    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.
**DocumentDecodeException**  
Exception defined in `pyglet.text`

```python
exception DocumentDecodeException
    An error occurred decoding document text.
```

**Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decode_attributed(text)</code></td>
<td>Create a document directly from some attributed text.</td>
</tr>
<tr>
<td><code>decode_html(text[, location=None])</code></td>
<td>Create a document directly from some HTML formatted text.</td>
</tr>
<tr>
<td><code>decode_text(text)</code></td>
<td>Create a document directly from some plain text.</td>
</tr>
<tr>
<td><code>get_decoder(filename[, mimetype])</code></td>
<td>Get a document decoder for the given filename and MIME type.</td>
</tr>
<tr>
<td><code>load(filename[, file, mimetype])</code></td>
<td>Load a document from a file.</td>
</tr>
</tbody>
</table>

**decode_attributed Function** Defined in `pyglet.text`

```python
decode_attributed(text)
    Create a document directly from some attributed text.
```

Parameters  
- `text (str)` – Attributed text to decode.

Return type  `FormattedDocument`

**decode_html Function** Defined in `pyglet.text`

```python
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`

```python
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

```python
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

```python
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:

```python
from pyglet.window import Window
win = Window(width=640, height=480)
```

Attach your own event handlers:

```python
@win.event
def on_key_press(symbol, modifiers):
    # ... handle this event ...
```

Place drawing code for the window within the `Window.on_draw` event handler:

```python
@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):

```python
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:

```python
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:

```python
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:

```python
from pyglet import gl
# Create template config
cfg = 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”):

cfgs = screen.get_matching_configs(config)
if not cfgs:
    # ... config is not supported
else:
    win = window.Window(config=cfgs[0])

Modules

event  Events for pyglet.window.
key    Key constants and utilities for pyglet.window.
mouse  Mouse constants and utilities for pyglet.window.

pyglet.window.event  Events for pyglet.window.
See Window for a description of the window event types.

WindowEventLogger  Print all events to a file.
WindowExitHandler  Determine if the window should be closed.

Classes

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())

Constructor:

    __init__ (logfile=None)
    Create a WindowEventLogger which writes to logfile.
Parameters **logfile** *(file-like object)* – The file to write to. If unspecified, stdout will be used.

Methods:

```python
on_activate()
on_close()
on_context_lost()
on_context_state_lost()
on_deactivate()
on_draw()
on_expose()
on_hide()
on_key_press(symbol, modifiers)
on_key_release(symbol, modifiers)
on_mouse_drag(x, y, dx, dy, buttons, modifiers)
on_mouse_enter(x, y)
on_mouse_leave(x, y)
on_mouse_motion(x, y, dx, dy)
on_mouse_press(x, y, button, modifiers)
on_mouse_release(x, y, button, modifiers)
on_mouse_scroll(x, y, dx, dy)
on_move(x, y)
on_resize(width, height)
on_show()
on_text(text)
on_text_motion(motion)
on_text_motion_select(motion)
```
WindowEventLogger.on_mouse_release(x, y, button, modifiers)
WindowEventLogger.on_mouse_scroll(x, y, dx, dy)
WindowEventLogger.on_move(x, y)
WindowEventLogger.on_resize(width, height)
WindowEventLogger.on_show()
WindowEventLogger.on_text(text)
WindowEventLogger.on_text_motion(motion)
WindowEventLogger.on_text_motion_select(motion)

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.

    Methods:
        on_close()
        on_key_press(symbol, modifiers)

    Attributes:
        has_exit

Methods
WindowExitHandler.on_close()
WindowExitHandler.on_key_press(symbol, modifiers)

Attributes
WindowExitHandler.has_exit = False
Defined

Notes  
• key  
• mouse  
• sys

**pyglet.window.key**  Key constants and utilities for pyglet.window.

Usage:

```python
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

**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:

    ```python
    >>> win = window.Window
    >>> keyboard = key.KeyStateHandler()
    ```
>>> win.push_handlers(keyboard)

# Hold down the "up" arrow...

>>> keyboard[key.UP]
True

>>> keyboard[key.DOWN]
False

Methods:

clear() -> None. Remove all items from D.
copy() -> a shallow copy of D
get((k,d]) -> D[k] if k in D, else d
has_key(k) -> True if D has a key k, else False
items() -> list of D's (key, value) pairs, ...
iteritems() -> an iterator over the (key, ...)
iterkeys() -> an iterator over the keys of D
itervalues(...)
keys() -> list of D's keys
on_key_press(symbol, modifiers)
on_key_release(symbol, modifiers)

pop((k,[d]) -> v, ...) If key is not found, d is returned if given, otherwise KeyError is raised
popitem() -> (k, v), ... 2-tuple; but raise KeyError if D is empty.
setdefault((k,d]) -> D.get(k,d), ...
update([E, ...]) If E present and has a .keys() method, does: for k in E: D[k] = E[k]
values() -> list of D's values
viewitems(...)
viewkeys(...)
viewvalues(...)

Methods

KeyStateHandler.on_key_press (symbol, modifiers)
KeyStateHandler.on_key_release (symbol, modifiers)

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.has_key(k) -> True if D has a key k, else False
KeyStateHandler.items () -> list of D’s (key, value) pairs, as 2-tuples
KeyStateHandler.iteritems () -> an iterator over the (key, value) items of D
KeyStateHandler.iterkeys () -> an iterator over the keys of D
KeyStateHandler.

**values** () → an iterator over the values of D

KeyStateHandler.

**keys** () → list of 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 present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, 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** () → list of D’s values

KeyStateHandler.

**viewitems** () → a set-like object providing a view on D’s items

KeyStateHandler.

**viewkeys** () → a set-like object providing a view on D’s keys

KeyStateHandler.

**viewvalues** () → an object providing a view on D’s values

---

### Functions

<table>
<thead>
<tr>
<th>modifier_string</th>
<th>Function</th>
<th>Defined in</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>modifiers_string</td>
<td>Defined in <code>pyglet.window.key</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**modifiers_string**(modifiers) → Return a string describing a set of modifiers.

Example:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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` = 'linux2'

  str(object='') -> string

  Return a nice string representation of the object. If the argument is a string, the return value is the same object.

**pyglet.window.mouse**  Mouse constants and utilities for pyglet.window.

**buttons_string** *(buttons)*

Return a string describing a set of active mouse buttons.

**Parameters**  `buttons` *(int)* – Bitwise combination of mouse button constants.

**Return type**  str

**Variables**

- `LEFT` = 1

  int(x=0) -> int or long int(x, base=10) -> int or long
Convert a number or string to an integer, or return 0 if no arguments are given. If \( x \) is floating point, the conversion truncates towards zero. If \( x \) is outside the integer range, the function returns a long instead.

If \( x \) is not a number or if base is given, then \( x \) must be a string or Unicode object 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. >>>

\[
\text{int('0b100', base=0)} \quad 4
\]

Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefaultMouseCursor</td>
<td>The default mouse cursor used by the operating system.</td>
</tr>
<tr>
<td>Display</td>
<td>A display device supporting one or more screens.</td>
</tr>
<tr>
<td>FPSDisplay</td>
<td>Display of a window’s framerate.</td>
</tr>
<tr>
<td>ImageMouseCursor</td>
<td>A user-defined mouse cursor created from an image.</td>
</tr>
<tr>
<td>MouseCursor</td>
<td>An abstract mouse cursor.</td>
</tr>
<tr>
<td>Platform</td>
<td>Operating-system-level functionality.</td>
</tr>
<tr>
<td>Window</td>
<td>Platform-independent application window.</td>
</tr>
</tbody>
</table>

**DefaultMouseCursor Class**

class **DefaultMouseCursor**

The default mouse cursor used by the operating system.

**Methods:**

```
draw(x, y)  # Abstract render method.
```
Attributes:

drawable

Attributes

DefaultMouseCursor.drawable = False

Inherited members

Methods

DefaultMouseCursor.draw(x, y)
Abstract render method.

The cursor should be drawn with the “hot” spot at the given coordinates. The projection is set to the pyglet default (i.e., orthographic in window-space), however no other aspects of the state can be assumed.

Parameters

• x (int) – X coordinate of the mouse pointer’s hot spot.
• y (int) – Y coordinate of the mouse pointer’s hot spot.

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.

Constructor:
__init__()

Methods:

get_default_screen() Get the default screen as specified by the user’s operating system preferences.
get_screens() Get the available screens.
get_windows() Get the windows currently attached to this display.
Methods

Display.get_default_screen()
Get the default screen as specified by the user’s operating system preferences.

Return type Screen

Display.get_screens()
Get the available screens.

A typical multi-monitor workstation comprises one Display with multiple Screen s. This method returns a list of screens which can be enumerated to select one for full-screen display.

For the purposes of creating an OpenGL config, the default screen will suffice.

Return type list of Screen

Display.get_windows()
Get the windows currently attached to this display.

Return type sequence of Window

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:

```python
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.

Constructor:

__init__ (window)

Methods:

```python
draw() Draw the label.
```
<table>
<thead>
<tr>
<th>set_fps(fps)</th>
<th>Set the label text for the given FPS estimation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>update()</td>
<td>Records a new data point at the current time.</td>
</tr>
</tbody>
</table>

**Attributes:**

- `update_period` : Time in seconds between updates.

**Methods**

- **FPSDisplay.draw()**
  
  Draw the label.
  
  The OpenGL state is assumed to be at default values, except that the MODELVIEW and PROJECTION matrices are ignored. At the return of this method the matrix mode will be MODELVIEW.

- **FPSDisplay.set_fps(fps)**
  
  Set the label text for the given FPS estimation.
  
  Called by `update` every `update_period` seconds.

  **Parameters**
  
  `- fps (float)` – Estimated framerate of the window.

- **FPSDisplay.update()**
  
  Records a new data point at the current time. This method is called automatically when the window buffer is flipped.

**Attributes**

- **FPSDisplay.update_period = 0.25**
  
  Time in seconds between updates.

  **Type**
  
  `float`

### 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.

**Constructor:**

- **__init__(image, hot_x=0, hot_y=0)**
  
  Create a mouse cursor from an image.
Parameters

• **image** (*pyglet.image.AbstractImage*) – Image to use for the mouse cursor. It must have a valid `texture` attribute.
• **hot_x** (*int*) – X coordinate of the “hot” spot in the image relative to the image’s anchor.
• **hot_y** (*int*) – Y coordinate of the “hot” spot in the image, relative to the image’s anchor.

Methods:

```python
draw(x, y)
```

Attributes:

```python
drawable
```

Methods

```python
ImageMouseCursor.draw(x, y)
```

Attributes

```python
ImageMouseCursor.drawable = True
```

**MouseCursor Class**

class **MouseCursor**

An abstract mouse cursor.

Methods:

```python
draw(x, y)  # Abstract render method.
```

Attributes:

```python
drawable  # Indicates if the cursor is drawn using OpenGL.
```

Methods
MouseCursor\texttt{.draw}(x, y)
Abstract render method.

The cursor should be drawn with the “hot” spot at the given coordinates. The projection is set to the pyglet default (i.e., orthographic in window-space), however no other aspects of the state can be assumed.

**Parameters**

- \texttt{x (int)} – X coordinate of the mouse pointer’s hot spot.
- \texttt{y (int)} – Y coordinate of the mouse pointer’s hot spot.

**Attributes**

\texttt{MouseCursor.drawable = True}
Indicates if the cursor is drawn using OpenGL. This is True for all mouse cursors except system cursors.

**Platform Class**

class \texttt{Platform}
Operating-system-level functionality.

The platform instance can only be obtained with \texttt{get\_platform}. Use the platform to obtain a \texttt{Display} instance.

| Warning: Deprecated. Use pyglet.canvas.Display |

**Methods:**

- \texttt{get\_default\_display()} Get the default display device.
- \texttt{get\_display(name)} Get a display device by name.

**Methods**

\texttt{Platform.get\_default\_display()}
Get the default display device.

| Warning: Deprecated. Use pyglet.canvas.get\_display. |

\texttt{Platform.get\_display(name)}
Get a display device by name.

This is meaningful only under X11, where the \texttt{name} is a string including the host name and display number; for example "localhost:1".

On platforms other than X11, \texttt{name} is ignored and the default display is returned. pyglet does not support multiple multiple video devices on Windows or OS X. If more than one device is attached, they will appear as a single virtual device comprising all the attached screens.
Warning: Deprecated. Use pyglet.canvas.get_display.

Parameters

name (str) – The name of the display to connect to.

Return type Display

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.

Constructor:

__init__ (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)

Create a window.

All parameters are optional, and reasonable defaults are assumed where they are not specified.

The display, screen, config and context parameters form a hierarchy of control: there is no need to specify more than one of these. For example, if you specify screen the display will be inferred, and a default config and context will be created.

config is a special case; it can be a template created by the user specifying the attributes desired, or it can be a complete config as returned from Screen.get_matching_configs or similar.

The context will be active as soon as the window is created, as if switch_to was just called.

Parameters
• **width** *(int)* – Width of the window, in pixels. Defaults to 640, or the screen width if `fullscreen` is True.

• **height** *(int)* – Height of the window, in pixels. Defaults to 480, or the screen height if `fullscreen` is True.

• **caption** *(str or unicode)* – Initial caption (title) of the window. Defaults to `sys.argv[0]`.

• **resizable** *(bool)* – If True, the window will be resizable. Defaults to False.

• **style** *(int)* – One of the `WINDOW_STYLE_*` constants specifying the border style of the window.

• **fullscreen** *(bool)* – If True, the window will cover the entire screen rather than floating. Defaults to False.

• **visible** *(bool)* – Determines if the window is visible immediately after creation. Defaults to True. Set this to False if you would like to change attributes of the window before having it appear to the user.

• **vsync** *(bool)* – If True, buffer flips are synchronised to the primary screen’s vertical retrace, eliminating flicker.

• **display** *(Display)* – The display device to use. Useful only under X11.

• **screen** *(Screen)* – The screen to use, if in fullscreen.

• **config** *(pyglet.gl.Config)* – Either a template from which to create a complete config, or a complete config.

• **context** *(pyglet.gl.Context)* – The context to attach to this window. The context must not already be attached to another window.

• **mode** *(ScreenMode)* – The screen will be switched to this mode if `fullscreen` is True. If None, an appropriate mode is selected to accommodate `width` and `height`.

### Methods:

- `activate()` – Attempt to restore keyboard focus to the window.
- `clear()` – Clear the window.
- `close()` – Close the window.
- `dispatch_event(*args)` – Poll the operating system event queue for new events and call attached event handlers.
- `dispatch_events()` – Draw the custom mouse cursor.
- `flip()` – Swap the OpenGL front and back buffers.
- `get_location()` – Return the current position of the window.
- `get_size()` – Return the current size of the window.
- `get_system_mouse_cursor(name)` – Obtain a system mouse cursor.
- `maximize()` – Maximize the window.
- `minimize()` – Minimize the window.
- `set_caption(caption)` – Set the window’s caption.
- `set_exclusive_keyboard([exclusive])` – Prevent the user from switching away from this window using keyboard accelerators.
- `set_exclusive_mouse([exclusive])` – Hide the mouse cursor and direct all mouse events to this window.
- `set_fullscreen([fullscreen, screen, mode, ...])` – Toggle to or from fullscreen.
- `set_icon(*images)` – Set the window icon.
- `set_location(x, y)` – Set the position of the window.
- `set_maximum_size(width, height)` – Set the maximum size of the window.
- `set_minimum_size(width, height)` – Set the minimum size of the window.
### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_mouse_cursor(cursor)</code></td>
<td>Change the appearance of the mouse cursor.</td>
</tr>
<tr>
<td><code>set_mouse_platform_visible(platform_visible)</code></td>
<td>Set the platform-drawn mouse cursor visibility.</td>
</tr>
<tr>
<td><code>set_mouse_visible(visible)</code></td>
<td>Show or hide the mouse cursor.</td>
</tr>
<tr>
<td><code>set_size(width, height)</code></td>
<td>Resize the window.</td>
</tr>
<tr>
<td><code>set_visible(visible)</code></td>
<td>Show or hide the window.</td>
</tr>
<tr>
<td><code>set_vsync(vsync)</code></td>
<td>Enable or disable vertical sync control.</td>
</tr>
<tr>
<td><code>switch_to()</code></td>
<td>Make this window the current OpenGL rendering context.</td>
</tr>
</tbody>
</table>

### Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on_activate()</code></td>
<td>The window was activated.</td>
</tr>
<tr>
<td><code>on_close()</code></td>
<td>The user attempted to close the window.</td>
</tr>
<tr>
<td><code>on_context_lost()</code></td>
<td>The window’s GL context was lost.</td>
</tr>
<tr>
<td><code>on_context_state_lost()</code></td>
<td>The state of the window’s GL context was lost.</td>
</tr>
<tr>
<td><code>on_deactivate()</code></td>
<td>The window was deactivated.</td>
</tr>
<tr>
<td><code>on_draw()</code></td>
<td>The window contents must be redrawn.</td>
</tr>
<tr>
<td><code>on_expose()</code></td>
<td>A portion of the window needs to be redrawn.</td>
</tr>
<tr>
<td><code>on_hide()</code></td>
<td>The window was hidden.</td>
</tr>
<tr>
<td><code>on_key_press(symbol, modifiers)</code></td>
<td>A key on the keyboard was pressed (and held down).</td>
</tr>
<tr>
<td><code>on_key_release(symbol, modifiers)</code></td>
<td>A key on the keyboard was released.</td>
</tr>
<tr>
<td><code>on_mouse_drag(x, y, dx, dy, buttons, modifiers)</code></td>
<td>The mouse was moved with one or more mouse buttons pressed.</td>
</tr>
<tr>
<td><code>on_mouse_enter(x, y)</code></td>
<td>The mouse was moved into the window.</td>
</tr>
<tr>
<td><code>on_mouse_leave(x, y)</code></td>
<td>The mouse was moved outside of the window.</td>
</tr>
<tr>
<td><code>on_mouse_motion(x, y, dx, dy)</code></td>
<td>The mouse was moved with no buttons held down.</td>
</tr>
<tr>
<td><code>on_mouse_press(x, y, button, modifiers)</code></td>
<td>A mouse button was pressed (and held down).</td>
</tr>
<tr>
<td><code>on_mouse_release(x, y, button, modifiers)</code></td>
<td>A mouse button was released.</td>
</tr>
<tr>
<td><code>on_mouse_scroll(x, y, button, modifiers)</code></td>
<td>The mouse wheel was released.</td>
</tr>
<tr>
<td><code>on_move(x, y)</code></td>
<td>The window was moved.</td>
</tr>
<tr>
<td><code>on_resize(width, height)</code></td>
<td>The window was resized.</td>
</tr>
<tr>
<td><code>on_show()</code></td>
<td>The window was shown.</td>
</tr>
<tr>
<td><code>on_text(text)</code></td>
<td>The user input some text.</td>
</tr>
<tr>
<td><code>on_text_motion(motion)</code></td>
<td>The user moved the text input cursor.</td>
</tr>
<tr>
<td><code>on_text_motion_select(motion)</code></td>
<td>The user moved the text input cursor while extending the selection.</td>
</tr>
</tbody>
</table>

### Attributes

- `CURSOR_CROSSHAIR`
- `CURSOR_DEFAULT`
- `CURSOR_HAND`
- `CURSOR_HELP`
- `CURSOR_NO`
- `CURSOR_SIZE`
- `CURSOR_SIZE_DOWN`
- `CURSOR_SIZE_DOWN_LEFT`
- `CURSOR_SIZE_DOWN_RIGHT`
- `CURSOR_SIZE_LEFT`
- `CURSOR_SIZE_LEFT_RIGHT`
Table 2.334 – continued from previous page

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURSOR_SIZE_RIGHT</td>
<td>The window caption (title).</td>
</tr>
<tr>
<td>CURSOR_SIZE_UP</td>
<td>A GL config describing the context of this window.</td>
</tr>
<tr>
<td>CURSOR_SIZE_UP_DOWN</td>
<td>The OpenGL context attached to this window.</td>
</tr>
<tr>
<td>CURSOR_SIZE_UP_LEFT</td>
<td>The display this window belongs to.</td>
</tr>
<tr>
<td>CURSOR_SIZE_UP_RIGHT</td>
<td>True if the window is currently fullscreen.</td>
</tr>
<tr>
<td>CURSOR_WAIT</td>
<td>The height of the window, in pixels.</td>
</tr>
<tr>
<td>CURSOR_WAIT_ARROW</td>
<td>True if the window is resizable.</td>
</tr>
<tr>
<td>WINDOW_STYLE_BORDERLESS</td>
<td>The screen this window is fullscreen in.</td>
</tr>
<tr>
<td>WINDOW_STYLE_DEFAULT</td>
<td>The window style; one of the WINDOW_STYLE_* constants.</td>
</tr>
<tr>
<td>WINDOW_STYLE_DIALOG</td>
<td>True if the window is currently visible.</td>
</tr>
<tr>
<td>WINDOW_STYLE_TOOL</td>
<td>True if buffer flips are synchronised to the screen’s vertical retrace.</td>
</tr>
<tr>
<td>width</td>
<td>The width of the window, in pixels.</td>
</tr>
</tbody>
</table>

**Methods**

**Window.activate()**

Attempt to restore keyboard focus to the window.

Depending on the window manager or operating system, this may not be successful. For example, on Windows XP an application is not allowed to “steal” focus from another application. Instead, the window’s taskbar icon will flash, indicating it requires attention.

**Window.clear()**

Clear the window.

This is a convenience method for clearing the color and depth buffer. The window must be the active context (see switch_to).

**Window.close()**

Close the window.

After closing the window, the GL context will be invalid. The window instance cannot be reused once closed (see also set_visible).

The `pyglet.app.EventLoop.on_window_close` event is dispatched on `pyglet.app.event_loop` when this method is called.

**Window.dispatch_event(*args)**
Window.dispatch_events()  
Poll the operating system event queue for new events and call attached event handlers.  
This method is provided for legacy applications targeting pyglet 1.0, and advanced applications that must integrate their event loop into another framework.  
Typical applications should use pyglet.app.run.

Window.draw_mouse_cursor()  
Draw the custom mouse cursor.  
If the current mouse cursor has drawable set, this method is called before the buffers are flipped to render it.  
This method always leaves the GL_MODELVIEW matrix as current, regardless of what it was set to previously. No other GL state is affected.  
There is little need to override this method; instead, subclass MouseCursor and provide your own draw method.

Window.flip()  
Swap the OpenGL front and back buffers.  
Call this method on a double-buffered window to update the visible display with the back buffer. The contents of the back buffer is undefined after this operation.  
Windows are double-buffered by default. This method is called automatically by EventLoop after the on_draw event.

Window.get_location()  
Return the current position of the window.  
Return type (int, int)  
Returns The distances of the left and top edges from their respective edges on the virtual desktop, in pixels.

Window.get_size()  
Return the current size of the window.  
The window size does not include the border or title bar.  
Return type (int, int)  
Returns The width and height of the window, in pixels.

Window.get_system_mouse_cursor(name)  
Obtain a system mouse cursor.  
Use set_mouse_cursor to make the cursor returned by this method active. The names accepted by this method are the CURSOR_* constants defined on this class.  
Parameters name (str) – Name describing the mouse cursor to return. For example, CURSOR_WAIT, CURSOR_HELP, etc.  
Return type MouseCursor  
Returns A mouse cursor which can be used with set_mouse_cursor.

Window.maximize()  
Maximize the window.  
The behaviour of this method is somewhat dependent on the user’s display setup. On a multi-monitor system, the window may maximize to either a single screen or the entire virtual desktop.

Window.minimize()  
Minimize the window.
Window.set_caption(caption)
Set the window's caption.

The caption appears in the titlebar of the window, if it has one, and in the taskbar on Windows and many X11 window managers.

Parameters caption (str or unicode) – The caption to set.

Window.set_exclusive_keyboard(exclusive=True)
Prevent the user from switching away from this window using keyboard accelerators.

When enabled, this feature disables certain operating-system specific key combinations such as Alt+Tab (Command+Tab on OS X). This can be useful in certain kiosk applications, it should be avoided in general applications or games.

Parameters exclusive (bool) – If True, exclusive keyboard is enabled, otherwise it is disabled.

Window.set_exclusive_mouse(exclusive=True)
Hide the mouse cursor and direct all mouse events to this window.

When enabled, this feature prevents the mouse leaving the window. It is useful for certain styles of games that require complete control of the mouse. The position of the mouse as reported in subsequent events is meaningless when exclusive mouse is enabled; you should only use the relative motion parameters dx and dy.

Parameters exclusive (bool) – If True, exclusive mouse is enabled, otherwise it is disabled.

Window.set_fullscreen(fullscreen=True, screen=None, mode=None, width=None, height=None)
Toggle to or from fullscreen.

After toggling fullscreen, the GL context should have retained its state and objects, however the buffers will need to be cleared and redrawn.

If width and height are specified and fullscreen is True, the screen may be switched to a different resolution that most closely matches the given size. If the resolution doesn’t match exactly, a higher resolution is selected and the window will be centered within a black border covering the rest of the screen.

Parameters

• fullscreen (bool) – True if the window should be made fullscreen, False if it should be windowed.

• screen (Screen) – If not None and fullscreen is True, the window is moved to the given screen. The screen must belong to the same display as the window.

• mode (ScreenMode) – The screen will be switched to the given mode. The mode must have been obtained by enumerating Screen.get_modes. If None, an appropriate mode will be selected from the given width and height.

• width (int) – Optional width of the window. If unspecified, defaults to the previous window size when windowed, or the screen size if fullscreen. Since: pyglet 1.2

• height (int) – Optional height of the window. If unspecified, defaults to the previous window size when windowed, or the screen size if fullscreen. Since: pyglet 1.2

Window.set_icon(*images)
Set the window icon.

If multiple images are provided, one with an appropriate size will be selected (if the correct size is not provided, the image will be scaled).

Useful sizes to provide are 16x16, 32x32, 64x64 (Mac only) and 128x128 (Mac only).

Parameters images (sequence of pyglet.image.AbstractImage) – List of images to use for the window icon.
Window.set_location(x, y)
Set the position of the window.

Parameters

• **x** (*int*) – Distance of the left edge of the window from the left edge of the virtual desktop, in pixels.

• **y** (*int*) – Distance of the top edge of the window from the top edge of the virtual desktop, in pixels.

Window.set_maximum_size(width, height)
Set the maximum size of the window.

Once set, the user will not be able to resize the window larger than the given dimensions. There is no way to remove the maximum size constraint on a window (but you could set it to a large value).

The behaviour is undefined if the maximum size is set smaller than the current size of the window.

The window size does not include the border or title bar.

Parameters

• **width** (*int*) – Maximum width of the window, in pixels.

• **height** (*int*) – Maximum height of the window, in pixels.

Window.set_minimum_size(width, height)
Set the minimum size of the window.

Once set, the user will not be able to resize the window smaller than the given dimensions. There is no way to remove the minimum size constraint on a window (but you could set it to 0,0).

The behaviour is undefined if the minimum size is set larger than the current size of the window.

The window size does not include the border or title bar.

Parameters

• **width** (*int*) – Minimum width of the window, in pixels.

• **height** (*int*) – Minimum height of the window, in pixels.

Window.set_mouse_cursor(cursor=None)
Change the appearance of the mouse cursor.

The appearance of the mouse cursor is only changed while it is within this window.

Parameters **cursor** (*MouseCursor*) – The cursor to set, or None to restore the default cursor.

Window.set_mouse_platform_visible(platform_visible=None)
Set the platform-drawn mouse cursor visibility. This is called automatically after changing the mouse cursor or exclusive mode.

Applications should not normally need to call this method, see set_mouse_visible instead.

Parameters **platform_visible** (*bool* or None) – If None, sets platform visibility to the required visibility for the current exclusive mode and cursor type. Otherwise, a bool value will override and force a visibility.

Window.set_mouse_visible(visibility=True)
Show or hide the mouse cursor.

The mouse cursor will only be hidden while it is positioned within this window. Mouse events will still be processed as usual.

Parameters **visible** (*bool*) – If True, the mouse cursor will be visible, otherwise it will be hidden.
Window.set_size(width, height)

Resize the window.

The behaviour is undefined if the window is not resizable, or if it is currently fullscreen.

The window size does not include the border or title bar.

Parameters

- width (int) – New width of the window, in pixels.
- height (int) – New height of the window, in pixels.

Window.set_visible(visible=True)

Show or hide the window.

Parameters visible (bool) – If True, the window will be shown; otherwise it will be hidden.

Window.set_vsync(vsync)

Enable or disable vertical sync control.

When enabled, this option ensures flips from the back to the front buffer are performed only during the vertical retrace period of the primary display. This can prevent “tearing” or flickering when the buffer is updated in the middle of a video scan.

Note that LCD monitors have an analogous time in which they are not reading from the video buffer; while it does not correspond to a vertical retrace it has the same effect.

With multi-monitor systems the secondary monitor cannot be synchronised to, so tearing and flicker cannot be avoided when the window is positioned outside of the primary display. In this case it may be advisable to forcibly reduce the framerate (for example, using pyglet.clock.set_fps_limit).

Parameters vsync (bool) – If True, vsync is enabled, otherwise it is disabled.

Window.switch_to()

Make this window the current OpenGL rendering context.

Only one OpenGL context can be active at a time. This method sets the current window’s context to be current. You should use this method in preference to pyglet.gl.Context.set_current, as it may perform additional initialisation functions.

Events

Window.on_activate()

The window was activated.

This event can be triggered by clicking on the title bar, bringing it to the foreground; or by some platform-specific method.

When a window is “active” it has the keyboard focus.

Window.on_close()

The user attempted to close the window.

This event can be triggered by clicking on the “X” control box in the window title bar, or by some other platform-dependent manner.

The default handler sets has_exit to True. In pyglet 1.1, if pyglet.app.event_loop is being used, close is also called, closing the window immediately.

Window.on_context_lost()

The window’s GL context was lost.
When the context is lost no more GL methods can be called until it is recreated. This is a rare event, triggered perhaps by the user switching to an incompatible video mode. When it occurs, an application will need to reload all objects (display lists, texture objects, shaders) as well as restore the GL state.

```python
Window.on_context_state_lost()
```

The state of the window’s GL context was lost.

Pyglet may sometimes need to recreate the window’s GL context if the window is moved to another video device, or between full-screen or windowed mode. In this case it will try to share the objects (display lists, texture objects, shaders) between the old and new contexts. If this is possible, only the current state of the GL context is lost, and the application should simply restore state.

```python
Window.on_deactivate()
```

The window was deactivated.

This event can be triggered by clicking on another application window. When a window is deactivated it no longer has the keyboard focus.

```python
Window.on_draw()
```

The window contents must be redrawn.

The `EventLoop` will dispatch this event when the window should be redrawn. This will happen during idle time after any window events and after any scheduled functions were called.

The window will already have the GL context, so there is no need to call `switch_to`. The window’s `flip` method will be called after this event, so your event handler should not.

You should make no assumptions about the window contents when this event is triggered; a resize or expose event may have invalidated the framebuffer since the last time it was drawn.

**Note:** Since pyglet 1.1

```python
Window.onExpose()
```

A portion of the window needs to be redrawn.

This event is triggered when the window first appears, and any time the contents of the window is invalidated due to another window obscuring it.

There is no way to determine which portion of the window needs redrawing. Note that the use of this method is becoming increasingly uncommon, as newer window managers composite windows automatically and keep a backing store of the window contents.

```python
Window.on_hide()
```

The window was hidden.

This event is triggered when a window is minimised or (on Mac OS X) hidden by the user.

```python
Window.on_key_press(symbol, modifiers)
```

A key on the keyboard was pressed (and held down).

In pyglet 1.0 the default handler sets `has_exit` to `True` if the ESC key is pressed.

In pyglet 1.1 the default handler dispatches the `on_close` event if the ESC key is pressed.

**Parameters**

- `symbol (int)` – The key symbol pressed.
- `modifiers (int)` – Bitwise combination of the key modifiers active.

```python
Window.on_key_release(symbol, modifiers)
```

A key on the keyboard was released.

**Parameters**
• `symbol (int)` – The key symbol pressed.
• `modifiers (int)` – Bitwise combination of the key modifiers active.

`Window.on_mouse_drag(x, y, dx, dy, buttons, modifiers)`

The mouse was moved with one or more mouse buttons pressed.

This event will continue to be fired even if the mouse leaves the window, so long as the drag buttons are continuously held down.

**Parameters**

• `x (int)` – Distance in pixels from the left edge of the window.
• `y (int)` – Distance in pixels from the bottom edge of the window.
• `dx (int)` – Relative X position from the previous mouse position.
• `dy (int)` – Relative Y position from the previous mouse position.
• `buttons (int)` – Bitwise combination of the mouse buttons currently pressed.
• `modifiers (int)` – Bitwise combination of any keyboard modifiers currently active.

`Window.on_mouse_enter(x, y)`

The mouse was moved into the window.

This event will not be trigged if the mouse is currently being dragged.

**Parameters**

• `x (int)` – Distance in pixels from the left edge of the window.
• `y (int)` – Distance in pixels from the bottom edge of the window.

`Window.on_mouse_leave(x, y)`

The mouse was moved outside of the window.

This event will not be trigged if the mouse is currently being dragged. Note that the coordinates of the mouse pointer will be outside of the window rectangle.

**Parameters**

• `x (int)` – Distance in pixels from the left edge of the window.
• `y (int)` – Distance in pixels from the bottom edge of the window.

`Window.on_mouse_motion(x, y, dx, dy)`

The mouse was moved with no buttons held down.

**Parameters**

• `x (int)` – Distance in pixels from the left edge of the window.
• `y (int)` – Distance in pixels from the bottom edge of the window.
• `dx (int)` – Relative X position from the previous mouse position.
• `dy (int)` – Relative Y position from the previous mouse position.

`Window.on_mouse_press(x, y, button, modifiers)`

A mouse button was pressed (and held down).

**Parameters**

• `x (int)` – Distance in pixels from the left edge of the window.
• `y (int)` – Distance in pixels from the bottom edge of the window.
• **button** (*int*) – The mouse button that was pressed.
• **modifiers** (*int*) – Bitwise combination of any keyboard modifiers currently active.

`Window.on_mouse_release(x, y, button, modifiers)`
A mouse button was released.

**Parameters**

• **x** (*int*) – Distance in pixels from the left edge of the window.
• **y** (*int*) – Distance in pixels from the bottom edge of the window.
• **button** (*int*) – The mouse button that was released.
• **modifiers** (*int*) – Bitwise combination of any keyboard modifiers currently active.

`Window.on_mouse_scroll(x, y, scroll_x, scroll_y)`
The mouse wheel was scrolled.

Note that most mice have only a vertical scroll wheel, so `scroll_x` is usually 0. An exception to this is the Apple Mighty Mouse, which has a mouse ball in place of the wheel which allows both `scroll_x` and `scroll_y` movement.

**Parameters**

• **x** (*int*) – Distance in pixels from the left edge of the window.
• **y** (*int*) – Distance in pixels from the bottom edge of the window.
• **scroll_x** (*int*) – Number of “clicks” towards the right (left if negative).
• **scroll_y** (*int*) – Number of “clicks” upwards (downwards if negative).

`Window.on_move(x, y)`
The window was moved.

**Parameters**

• **x** (*int*) – Distance from the left edge of the screen to the left edge of the window.
• **y** (*int*) – Distance from the top edge of the screen to the top edge of the window. Note that this is one of few methods in pyglet which use a Y-down coordinate system.

`Window.on_resize(width, height)`
The window was resized.

The window will have the GL context when this event is dispatched; there is no need to call `switch_to` in this handler.

**Parameters**

• **width** (*int*) – The new width of the window, in pixels.
• **height** (*int*) – The new height of the window, in pixels.

`Window.on_show()`
The window was shown.

This event is triggered when a window is restored after being minimised, or after being displayed for the first time.

`Window.on_text(text)`
The user input some text.

Typically this is called after `on_key_press` and before `on_key_release`, but may also be called multiple times if the key is held down (key repeating); or called without key presses if another input method was used (e.g., a pen input).
You should always use this method for interpreting text, as the key symbols often have complex mappings to their unicode representation which this event takes care of.

**Parameters** text *(unicode)* – The text entered by the user.

**Window.on_text_motion** *(motion)*

The user moved the text input cursor.

Typically this is called after `on_key_press` and before `on_key_release`, but may also be called multiple times if the key is held down (key repeating).

You should always use this method for moving the text input cursor (caret), as different platforms have different default keyboard mappings, and key repeats are handled correctly.

The values that `motion` can take are defined in `pyglet.window.key`:

- MOTION_UP
- MOTION_RIGHT
- MOTION_DOWN
- MOTION_LEFT
- MOTION_NEXT_WORD
- MOTION_PREVIOUS_WORD
- MOTION_BEGINNING_OF_LINE
- MOTION_END_OF_LINE
- MOTION_NEXT_PAGE
- MOTION_PREVIOUS_PAGE
- MOTION_BEGINNING_OF_FILE
- MOTION_END_OF_FILE
- MOTION_BACKSPACE
- MOTION_DELETE

**Parameters** motion *(int)* – The direction of motion; see remarks.

**Window.on_text_motion_select** *(motion)*

The user moved the text input cursor while extending the selection.

Typically this is called after `on_key_press` and before `on_key_release`, but may also be called multiple times if the key is held down (key repeating).

You should always use this method for responding to text selection events rather than the raw `on_key_press`, as different platforms have different default keyboard mappings, and key repeats are handled correctly.

The values that `motion` can take are defined in `pyglet.window.key`:

- MOTION_UP
- MOTION_RIGHT
- MOTION_DOWN
- MOTION_LEFT
- MOTION_NEXT_WORD
- MOTION_PREVIOUS_WORD
•MOTION_BEGINNING_OF_LINE
•MOTION_END_OF_LINE
•MOTION_NEXT_PAGE
•MOTION_PREVIOUS_PAGE
•MOTION_BEGINNING_OF_FILE
•MOTION_END_OF_FILE

Parameters **motion** *(int)* – The direction of selection motion; see remarks.

**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.captions**

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_press', ... 'on_move', 'on_activate', 'on_deactivate', 'on_show', 'on_hide', 'on_context_lost', 'on_context_state_lost', 'on_draw']`

`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.resizable`  
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

2.1. pyglet
Methods

Window.event(*args)
Function decorator for an event handler.

Usage:

```python
win = window.Window()

@win.event
def on_resize(self, width, height):
    # ...
```

or:

```python
@win.event('on_resize')
def foo(self, width, height):
    # ...
```

Window.pop_handlers()
Pop the top level of event handlers off the stack.

Window.push_handlers(*args, **kwargs)
Push a level onto the top of the handler stack, then attach zero or more event handlers.

If keyword arguments are given, they name the event type to attach. Otherwise, a callable’s __name__ attribute will be used. Any other object may also be specified, in which case it will be searched for callables with event names.

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.

Window.remove_handler(name, handler)
Remove a single event handler.

The given event handler is removed from the first handler stack frame it appears in. The handler must be the exact same callable as passed to set_handler, set_handlers or push_handlers; and the name must match the event type it is bound to.

No error is raised if the event handler is not set.

Parameters

- name (str) – Name of the event type to remove.
- handler (callable) – Event handler to remove.

Window.remove_handlers(*args, **kwargs)
Remove event handlers from the event stack.

See push_handlers for the accepted argument types. All handlers are removed from the first stack frame that contains any of the given handlers. No error is raised if any handler does not appear in that frame, or if no stack frame contains any of the given handlers.

If the stack frame is empty after removing the handlers, it is removed from the stack. Note that this interferes with the expected symmetry of push_handlers and pop_handlers.

Window.set_handler(name, handler)
Attach a single event handler.
Parameters

- **name** *(str)* – Name of the event type to attach to.
- **handler** *(callable)* – Event handler to attach.

```
Window.set_handlers(*args, **kwargs)
```

Attach one or more event handlers to the top level of the handler stack.

See `push_handlers` for the accepted argument types.

Exceptions

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MouseCursorException</td>
<td>The root exception for all mouse cursor-related errors.</td>
</tr>
<tr>
<td>NoSuchConfigException</td>
<td>An exception indicating the requested configuration is not available.</td>
</tr>
<tr>
<td>NoSuchDisplayException</td>
<td>An exception indicating the requested display is not available.</td>
</tr>
<tr>
<td>NoSuchScreenModeException</td>
<td>An exception indicating the requested screen resolution could not be met.</td>
</tr>
<tr>
<td>WindowException</td>
<td>The root exception for all window-related errors.</td>
</tr>
</tbody>
</table>

---

**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.
**NoSuchDisplayException**  Exception defined in `pyglet.window`  

**NoSuchDisplayException**  An exception indicating the requested display is not available.

\[
\text{pyglet.windowWindowException} \rightarrow \text{pyglet.windowNoSuchScreenModeException}
\]

**NoSuchScreenModeException**  Exception defined in `pyglet.window`  

**NoSuchScreenModeException**  An exception indicating the requested screen resolution could not be met.

**WindowException**  
Exception defined in `pyglet.window`  

**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.
Defined

- gl
- pprint
- pyglet
- sys
3.1 Importing pyglet

pyglet is a cross-platform games and multimedia package.

Detailed documentation is available at http://www.pyglet.org


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 resize 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** = ‘1.2.4’

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.:

```python
>>> import pyglet
>>> from pkg_resources import parse_version
>>> parse_version(pyglet.version) >= parse_version('1.1')
True
```

### 3.2 Advanced topics

- **Environment settings**

#### 3.2.1 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:
### 3.3 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.

#### 3.3.1 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__`ed.

   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.

#### 3.3.2 Missing extensions

The latest `glext.h` on opengl.org and nvidia 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.

---

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>pyglet.options key</th>
<th>Type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PYGLET_AUDIO</td>
<td>audio</td>
<td>List of strings</td>
<td><code>directsound,openal,alsa,silent</code></td>
</tr>
<tr>
<td>PYGLET_DEBUG_GLE</td>
<td>gldebug_gl</td>
<td>Boolean</td>
<td>1 (^1)</td>
</tr>
</tbody>
</table>

\(^1\) Defaults to 1 unless Python is run with `-O` or from a frozen executable.
3.4 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).

3.4.1 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
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
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`:
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
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.

### 3.4.2 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.

### 3.4.3 Generating Xlib wrappers

On Linux with the Xinerama extension installed (doesn’t have to be in use, just available), run:

```python
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.

### 3.5 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
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.

### 3.5.1 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.

### 3.5.2 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:

```python
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, variable 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:

```c
#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.

### 3.5.3 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 `TypeSpecifier`s; 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.

### 3.5.4 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.

### 3.6 Making a pyglet release

1. hg pull -u
2. Update version string in setup.py, pyglet/_init__.py and CHANGELOG Update README, and the two readme.rtf files on Windows and Mac.
3. hg push
4. Mac OS X release (requires OS X 10.5 and developer tools installed):

   ```bash
   sudo tools/genmpkg/genmpkg.sh
   ```

   Creates .dmg in dist/

   NOTE for 10.5: bdist_mpkg doesn’t quite work, needs a hack to avoid doing the admin write check (you’ll see when you get the traceback).

5. Windows release (requires WIX 3.0. WIX bin/ must be in PATH):

   ```bash
   python tools/genmsi/genmsi.py
   ```

   Creates .msi in dist/
6. Linux - You will need docutils, the docbook writer from the docutils sandbox, inkscape, fop, docbook-xsl, and perhaps more:

   ./make.py clean ; ./make.py docs

   Creates doc package in dist/

7. Source, egg and doc releases:

   tools/gendist.sh

   Creates .eggs, .tar.gzs and .zips in dist/

8. Upload files to googlecode:

   python tools/upload/upload.py


10. Copy URLs reported from upload into website/download.xml

11. Add news item to website/news-items.xml

12. Regenerate website with tools/genwebsite.sh

13. Update pyglet.org from website/dist/ to /:

   download.html, news.html, news.xml, index.html

   From doc to doc:

   html/api/, html/programming_guide/ pdf/programming_guide.pdf

14. python setup.py register

15. Tell people!

Untested AFAIK:

- Source distros on any platform
- Eggs on any platform: both to be installed and without installation (‘require’)
- Windows Vista
- Upgrade of Mac OS X install (after changing version string to beta, etc)
- Mac OS X with Python 2.4 + ctypes (ctypes didn’t compile on my mac).
- Mac OS X 10.3

### 3.7 Distribution

#### 3.7.1 Self-contained executables

Creating icons

Mac OS X requires several different icon images combined into a single .icns file. Use Icon Composer to create this file (this is installed in /Developer/Applications/Utilities with the Developer Tools).
3.8 Documentation

This is the pyglet documentation, generated with sphinx.

Details:

<table>
<thead>
<tr>
<th>Date</th>
<th>2015/09/02 04:29:47</th>
</tr>
</thead>
<tbody>
<tr>
<td>pyglet version</td>
<td>1.2.4</td>
</tr>
</tbody>
</table>

Note: See the Sphinx warnings log file for errors.

3.8.1 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

```python
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.
```

3.8. Documentation 459
```python
#: 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.

<table>
<thead>
<tr>
<th>Source</th>
<th>Shows</th>
</tr>
</thead>
<tbody>
<tr>
<td>:deprecated:</td>
<td><strong>Warning</strong>: Deprecated. Do not use</td>
</tr>
<tr>
<td>Do not use</td>
<td></td>
</tr>
<tr>
<td>:since:</td>
<td><strong>Note</strong>: Since 1.2</td>
</tr>
<tr>
<td>1.2</td>
<td></td>
</tr>
</tbody>
</table>
```

460 Chapter 3. Internals
Internal references

To cross-reference to any documented API member, use the following roles:

<table>
<thead>
<tr>
<th>Source</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>:mod:<code>pyglet.app</code></td>
<td>The <code>pyglet.app</code> module</td>
</tr>
<tr>
<td>:func:<code>~pyglet.app.run</code></td>
<td>The <code>run()</code> function</td>
</tr>
<tr>
<td>:class:<code>~pyglet.window.Window</code></td>
<td>The <code>Window</code> class</td>
</tr>
<tr>
<td>:meth:<code>~pyglet.window.Window.close</code></td>
<td>The <code>close()</code> method</td>
</tr>
<tr>
<td>:attr:<code>~pyglet.window.Window.fullscreen</code></td>
<td>The <code>fullscreen</code> attribute</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Source</th>
<th>Shows</th>
</tr>
</thead>
<tbody>
<tr>
<td>:guide:<code>doc_ref</code></td>
<td>See also: Programming Guide - Internal references</td>
</tr>
<tr>
<td>:ref:<code>My_Anchor&lt;my_anchor&gt;</code></td>
<td>MyAnchor</td>
</tr>
</tbody>
</table>

3.8.2 Generation

The complete documentation can be generated using `sphinx`.

Requirements

- `sphinx`: Documentation builder.
- `graphviz`: If you want to generate graphs.

To build the documentation, execute:

```
./make.py docs
```

If the build succeeds, the web pages are in `_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.

Omisions

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:

```python
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.font.carbon
- 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.avbin
- pyglet.media.drivers.directsound
- pyglet.media.drivers.openal
- pyglet.media.drivers.pulse
- 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_epydoc` is True.

**Known bugs**

- The Window class attributes are not documented because they are defined at BaseWindow class.

### 3.9 tests.test

Test framework for pyglet. Reads details of components and capabilities from a requirements document, runs the appropriate unit tests.
3.9.1 How to Run the Tests

```bash
python tests/test.py top app graphics clock resource # these all run automatically
python tests/test.py font media text
python tests/test.py image
python tests/test.py window
```

Because the tests are interactive, they can take quite a while to complete. The ‘window’ section in particular takes a long time. It can be frustrating to get almost through the tests and then something gets messed up, so we suggest you run the tests in sections as listed above. If you are curious, the sections are defined in tests/plan.txt.

Here are the different sections and how long they take.

<table>
<thead>
<tr>
<th>Section</th>
<th>Time to Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>automatic</td>
</tr>
<tr>
<td>app</td>
<td>automatic</td>
</tr>
<tr>
<td>graphics</td>
<td>automatic</td>
</tr>
<tr>
<td>clock</td>
<td>automatic</td>
</tr>
<tr>
<td>resource</td>
<td>automatic</td>
</tr>
<tr>
<td>font</td>
<td>1 minute</td>
</tr>
<tr>
<td>media</td>
<td>1 minute</td>
</tr>
<tr>
<td>text</td>
<td>1 minute</td>
</tr>
<tr>
<td>image</td>
<td>5 minutes</td>
</tr>
<tr>
<td>window</td>
<td>10 minutes</td>
</tr>
</tbody>
</table>

3.9.2 Overview

First, some definitions:

**Test case:** A single test, implemented by a Python module in the tests/ directory. Tests can be interactive (requiring the user to pass or fail them) or non-interactive (the test passes or fails itself).

**Section:** A list of test cases to be run in a specified order. Sections can also contain other sections to an arbitrary level.

**Capability:** A capability is a tag that can be applied to a test-case, which specifies a particular instance of the test. The tester can select which capabilities are present on their system; and only test cases matching those capabilities will be run.

There are platform capabilities “WIN”, “OSX” and “X11”, which are automatically selected by default.

The “DEVELOPER” capability is used to mark test cases which test a feature under active development.

The “GENERIC” capability signifies that the test case is equivalent under all platforms, and is selected by default.

Other capabilities can be specified and selected as needed. For example, we may wish to use an “NVIDIA” or “ATI” capability to specialise a test-case for a particular video card make.

Some tests generate regression images if enabled, so you will only need to run through the interactive procedure once. During subsequent runs the image shown on screen will be compared with the regression images and passed automatically if they match. There are command line options for enabling this feature.

By default regression images are saved in tests/regression/images/

3.9.3 Running tests

The test procedure is interactive (this is necessary to facilitate the many GUI-related tests, which cannot be completely automated). With no command-line arguments, all test cases in all sections will be run:
Before each test, a description of the test will be printed, including some information of what you should look for, and
what interactivityLiteral block is provided (including how to stop the test). Press ENTER to begin the test.

When the test is complete, assuming there were no detectable errors (for example, failed assertions or an exception),
you will be asked to enter a [P]ass or [F]ail. You should Fail the test if the behaviour was not as described, and enter a
short reason.

Details of each test session are logged for future use.

Command-line options:

--plan= Specify the test plan file (defaults to tests/plan.txt)

--test-root= Specify the top-level directory to look for unit tests in (defaults to test/)

--capabilities= Specify the capabilities to select, comma separated. By default this only includes your operating
system capability (X11, WIN or OSX) and GENERIC.

--log-level= Specify the minimum log level to write (defaults to 20: info)

--log-file= Specify log file to write to (defaults to “pyglet.%d.log”)

--regression-capture Save regression images to disk. Use this only if the tests have already been shown to pass.

--regression-check Look for a regression image on disk instead of prompting the user for passage. If a regression
image is found, it is compared with the test case using the tolerance specified below. Recommended only for
developers.

--regression-tolerance= Specify the tolerance when comparing a regression image. A value of 2, for example, means
each sample component must be +/- 2 units of the regression image. Tolerance of 0 means images must be
identical, tolerance of 256 means images will always match (if correct dimensions). Defaults to 2.

--regression-path= Specify the directory to store and look for regression images. Defaults to tests/regression/images/

--developer Selects the DEVELOPER capability.

--no-interactive= Don’t write descriptions or prompt for confirmation; just run each test in succession.

After the command line options, you can specify a list of sections or test cases to run.

3.9.4 Examples

python tests/test.py --capabilities=GENERIC,NVIDIA,WIN window

Runs all tests in the window section with the given capabilities. Test just the FULLSCREEN_TOGGLE test case
without prompting for input (useful for development).

python tests/image/PIL_RGBA_SAVE.py

Run a single test outside of the test harness. Handy for development; it is equivalent to specifying --no-interactive.

3.9.5 Writing tests

Add the test case to the appropriate section in the test plan (plan.txt). Create one unit test script per test case. For
example, the test for window.FULLSCREEN_TOGGLE is located at:

tests/window/FULLSCREEN_TOGGLE.py

The test file must contain:
• A module docstring describing what the test does and what the user should look for.
• One or more subclasses of unittest.TestCase.
• No other module-level code, except perhaps an if __name__ == '__main__' condition for running tests stand-alone.
• Optionally, the attribute '__noninteractive = True' to specify that the test is not interactive; doesn’t require user intervention.

During development, test cases should be marked with DEVELOPER. Once finished add the WIN, OSX and X11 capabilities, or GENERIC if it’s platform independent.

### 3.9.6 Writing regression tests

Your test case should subclass tests.regression.ImageRegressionTestCase instead of unittest.TestCase. At the point where the buffer (window image) should be checked/saved, call self.capture_regression_image(). If this method returns True, you can exit straight away (regression test passed), otherwise continue running interactively (regression image was captured, wait for user confirmation). You can call capture_regression_image() several times; only the final image will be used.

### 3.9.7 Python 3

The tests have to be processed by 2to3 in order to run them with Python 3.

This can be done with:

```
2to3 --output-dir=tests3 -W -n tests
```

And then run the tests int tests3 directory.
Related Documentation

- OpenGL Programming Guide
- OpenGL Reference Pages
- AVbin Documentation
- ctypes Reference
- Python Documentation
<table>
<thead>
<tr>
<th>Character</th>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>pyglet.app</td>
<td>87</td>
</tr>
<tr>
<td>c</td>
<td>pyglet.canvas</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>pyglet.clock</td>
<td>91</td>
</tr>
<tr>
<td>e</td>
<td>pyglet.event</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>pyglet.extlibs</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>pyglet.extlibs.png</td>
<td>106</td>
</tr>
<tr>
<td>f</td>
<td>pyglet.font</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>pyglet.font.fontconfig</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>pyglet.font.ttf</td>
<td>128</td>
</tr>
<tr>
<td>g</td>
<td>pyglet.gl</td>
<td>137</td>
</tr>
<tr>
<td></td>
<td>pyglet.gl.gl</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>pyglet.gl.gl_info</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>pyglet.gl.glu</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>pyglet.gl.glu_info</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>pyglet.gl.lib</td>
<td>144</td>
</tr>
<tr>
<td></td>
<td>pyglet.graphics</td>
<td>148</td>
</tr>
<tr>
<td></td>
<td>pyglet.graphics.allocation</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>pyglet.graphics.vertexattribute</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td>pyglet.graphics.vertexbuffer</td>
<td>167</td>
</tr>
<tr>
<td></td>
<td>pyglet.graphics.vertexdomain</td>
<td>177</td>
</tr>
<tr>
<td>i</td>
<td>pyglet.image</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td>pyglet.image.atlas</td>
<td>193</td>
</tr>
<tr>
<td></td>
<td>pyglet.image.codecs</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>pyglet.image.codecs.bmp</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>pyglet.image.codecs.dds</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>pyglet.image.codecs.gif</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>pyglet.image.codecs.png</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>pyglet.image.codecs.s3tc</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>pyglet.info</td>
<td>264</td>
</tr>
<tr>
<td></td>
<td>pyglet.input</td>
<td>265</td>
</tr>
<tr>
<td></td>
<td>pyglet.input.evdev_constants</td>
<td>266</td>
</tr>
<tr>
<td>m</td>
<td>pyglet.media</td>
<td>268</td>
</tr>
<tr>
<td></td>
<td>pyglet.media.drivers</td>
<td>268</td>
</tr>
<tr>
<td></td>
<td>pyglet.media.drivers.silent</td>
<td>268</td>
</tr>
<tr>
<td></td>
<td>pyglet.media.procedural</td>
<td>272</td>
</tr>
<tr>
<td></td>
<td>pyglet.media.riff</td>
<td>283</td>
</tr>
<tr>
<td>p</td>
<td>pyglet</td>
<td>449</td>
</tr>
<tr>
<td>r</td>
<td>pyglet.resource</td>
<td>322</td>
</tr>
<tr>
<td>s</td>
<td>pyglet.sprite</td>
<td>333</td>
</tr>
<tr>
<td>t</td>
<td>pyglet.text</td>
<td>340</td>
</tr>
<tr>
<td></td>
<td>pyglet.text.caret</td>
<td>341</td>
</tr>
<tr>
<td></td>
<td>pyglet.text.document</td>
<td>345</td>
</tr>
<tr>
<td></td>
<td>pyglet.text.formats</td>
<td>363</td>
</tr>
<tr>
<td></td>
<td>pyglet.text.formats.attributed</td>
<td>363</td>
</tr>
<tr>
<td></td>
<td>pyglet.text.formats.html</td>
<td>364</td>
</tr>
<tr>
<td></td>
<td>pyglet.text.formats.plaintext</td>
<td>367</td>
</tr>
<tr>
<td></td>
<td>pyglet.text.formats.structured</td>
<td>368</td>
</tr>
<tr>
<td></td>
<td>pyglet.text.layout</td>
<td>373</td>
</tr>
<tr>
<td></td>
<td>pyglet.text.runlist</td>
<td>393</td>
</tr>
<tr>
<td></td>
<td>tests.test</td>
<td>464</td>
</tr>
<tr>
<td>W</td>
<td>pyglet.window</td>
<td>415</td>
</tr>
<tr>
<td></td>
<td>pyglet.window.event</td>
<td>417</td>
</tr>
<tr>
<td></td>
<td>pyglet.window.key</td>
<td>420</td>
</tr>
<tr>
<td></td>
<td>pyglet.window.mouse</td>
<td>423</td>
</tr>
</tbody>
</table>
Symbols

__init__() (AbstractAttribute method), 154
__init__() (AbstractAudioPlayer method), 293
__init__() (AbstractDocument method), 347
__init__() (AbstractImage method), 202
__init__() (Allocator method), 150, 194
__init__() (Animation method), 205
__init__() (AnimationFrame method), 207
__init__() (AudioData method), 296
__init__() (AudioFormat method), 297
__init__() (Batch method), 184
__init__() (BufferImage method), 207
__init__() (BufferImageMask method), 210
__init__() (BufferManager method), 213
__init__() (Caret method), 342
__init__() (CheckerImagePattern method), 214
__init__() (Clock method), 93
__init__() (ClockDisplay method), 97
__init__() (ColorAttribute method), 155
__init__() (ColorBufferImage method), 215
__init__() (CompressedImageData method), 217
__init__() (ConstRunIterator method), 395
__init__() (DepthBufferImage method), 220
__init__() (DepthTexture method), 222
__init__() (Display method), 425
__init__() (DocumentLabel method), 400
__init__() (EdgeFlagAttribute method), 156
__init__() (FPSDisplay method), 426
__init__() (FileLocation method), 323
__init__() (FilteredRunIterator method), 395
__init__() (FogCoordAttribute method), 158
__init__() (FontConfig method), 121
__init__() (FontConfigPattern method), 122
__init__() (FontConfigSearchPattern method), 122
__init__() (FontConfigSearchResult method), 123
__init__() (FormattedDocument method), 352
__init__() (GenericAttribute method), 159
__init__() (GlyphString method), 131
__init__() (Group method), 186
__init__() (HTMLDecoder method), 365
__init__() (HTMLLabel method), 404
__init__() (Image method), 107
__init__() (ImageData method), 226
__init__() (ImageDataRegion method), 230
__init__() (ImageElement method), 368
__init__() (ImageGrid method), 234
__init__() (ImageMouseCursor method), 427
__init__() (IncrementalTextLayout method), 375
__init__() (IndexedVertexDomain method), 178
__init__() (IndexedVertexList method), 179
__init__() (IndirectArrayRegion method), 171
__init__() (InlineElement method), 357
__init__() (Label method), 409
__init__() (Loader method), 324
__init__() (ManagedSoundPlayer method), 297
__init__() (MappableVertexBufferObject method), 172
__init__() (MediaEvent method), 301
__init__() (MediaThread method), 301
__init__() (MultiTexCoordAttribute method), 160
__init__() (NormalAttribute method), 161
__init__() (NullGroup method), 187
__init__() (OrderedGroup method), 188
__init__() (OrderedListBuilder method), 370
__init__() (OverriddenRunIterator method), 396
__init__() (Player method), 302
__init__() (PlayerGroup method), 307
__init__() (ProceduralSource method), 273
__init__() (RIFFChunk method), 283
__init__() (RIFFFile method), 284
__init__() (RIFFForm method), 285
__init__() (RIFFTType method), 285
__init__() (Reader method), 108
__init__() (RunIterator method), 397
__init__() (RunList method), 397
__init__() (Saw method), 275
__init__() (ScrollableTextLayout method), 381
__init__() (ScrollableTextLayoutGroup method), 384
__init__() (SecondaryColorAttribute method), 162
__init__() (Silence method), 276
__init__() (SilentAudioPacket method), 269

Index
asRGB() (Reader method), 109
asRGB8() (Reader method), 109
asRGBA() (Reader method), 109
asRGBA8() (Reader method), 109
attributed (in module pyglet.resource), 330
attributed() (Loader method), 325
AttributedTextDecoder (class in pyglet.text.formats.attributed), 364
audio_format (Source attribute), 309
AudioData (class in pyglet.media), 296
AudioFormat (class in pyglet.media), 296
author (SourceInfo attribute), 312
AVbinSourceLoader (class in pyglet.media), 292

B
background_group (TextLayout attribute), 388
BASELINE (Text attribute), 134
Batch (class in pyglet.graphics), 184
batch (Sprite attribute), 335
begin() (ListBuilder method), 369
begin_update() (TextLayout method), 387
bind() (AbstractBuffer method), 168
bind() (MappableVertexBufferObject method), 172
bind() (VertexArray method), 173
bind() (VertexBufferObject method), 175
blit() (AbstractImage method), 202
blit() (ImageData method), 227
blit() (Texture method), 239
blit_into() (AbstractImage method), 202
blit_into() (DepthTexture method), 223
blit_into() (Texture method), 239
blit_into() (TextureRegion method), 252
blit_tiled() (TiledTexture method), 257
blit_to_texture() (AbstractImage method), 202
blit_to_texture() (ColorBufferImage method), 215
blit_to_texture() (CompressedImageData method), 218
blit_to_texture() (DepthBufferImage method), 220
blit_to_texture() (ImageData method), 227
bold (DocumentLabel attribute), 401
bold (FontConfigSearchResult attribute), 123
BOTTOM (Text attribute), 134
BufferImage (class in pyglet.image), 207
BufferImageMask (class in pyglet.image), 210
BufferManager (class in pyglet.image), 213
buttons_string() (in module pyglet.window.mouse), 423

C
c_void (class in pyglet.gl.lib), 145
call_scheduled_functions() (Clock method), 94
CannotSeekException, 320
caption (Window attribute), 442
Caret (class in pyglet.text.caret), 341
CENTER (Text attribute), 134
c_char_index() (FontConfig method), 121
check_bitdepth_color_type() (in module pyglet.extlibs.png), 115
check_color() (module pyglet.extlibs.png), 116
check_palette() (module pyglet.extlibs.png), 116
check_sizes() (module pyglet.extlibs.png), 116
CheckerImagePattern (class in pyglet.image), 214
chunk() (Reader method), 110
ChunkError, 114
chunk_length_type() (Reader method), 110
chunks() (Reader method), 110
clear() (AbstractAudioPlayer method), 293
clear() (SilentAudioPlayerPacketConsumer method), 270
clear() (SilentTimeAudioPlayer method), 271
clear() (Window method), 433
clear_jobs() (WorkerThread method), 319
Clock (class in pyglet.clock), 93
ClockDisplay (class in pyglet.clock), 97
close() (TruetypeInfo method), 129
close() (Window method), 433
color (Caret attribute), 344
color (DocumentLabel attribute), 401
color (Sprite attribute), 336
color (Text attribute), 134
color_as_bytes() (module pyglet.image), 262
color_triple() (module pyglet.extlibs.png), 116
ColorAttribute (class in pyglet.graphics.vertexattribute), 155
ColorBufferImage (class in pyglet.image), 214
colors (VertexList attribute), 182
columns (TextureGrid attribute), 248
columns() (SourceInfo attribute), 312
comment (SourceInfo attribute), 312
compat_platform (module pyglet.app), 89
compat_platform (module pyglet.clock), 101
compat_platform (module pyglet.font), 136
compat_platform (module pyglet.gl), 147
compat_platform (module pyglet.graphics), 191
compat_platform (module pyglet.graphics.vertexattrib), 167
compat_platform (module pyglet.graphics.vertexbuffer), 177
compat_platform (module pyglet.graphics.vertexdomain), 183
compat_platform (module pyglet.image), 263
compat_platform (module pyglet.image.codecs), 200
compat_platform (module pyglet.sprite), 340
compat_platform (module pyglet.text.layout), 393
compat_platform (module pyglet.window), 423
CompressedImageData (class in pyglet.image), 217
cone_inner_angle (Player attribute), 304
cone_orientation (Player attribute), 304
cone_outer_angle (Player attribute), 304
cone_outer_gain (Player attribute), 304
config (Window attribute), 442
ConfigException, 147
ConstRunIterator (class in pyglet.text.runlist), 395
consume() (AudioData method), 296
consume() (SilentAudioPacket method), 270
content_valign (TextLayout attribute), 388
text (Window attribute), 443
ContextException, 147
convert_pnm() (Writer method), 113
convert_ppm_and_pgm() (Writer method), 113
crypt_to_multi_tex_coord_attribute() (TexCoordAttribute method), 164
copyright (SourceInfo attribute), 312
create() (in module pyglet.image), 262
create() (IndexedVertexDomain method), 178
create() (pyglet.image.Texture class method), 239
create() (VertexDomain method), 181
create_attribute() (in module pyglet.graphics.vertexattribute), 166
create_attribute_usage() (in module pyglet.graphics.vertexdomain), 183
create_audio_driver() (in module pyglet.media.drivers.silent), 272
create_audio_player() (AbstractAudioDriver method), 292
create_audio_player() (SilentAudioDriver method), 269
create_buffer() (in module pyglet.graphics.vertexbuffer), 176
create_domain() (in module pyglet.graphics.vertexdomain), 183
create_for_image() (pyglet.image.TileableTexture class method), 257
create_for_image_grid() (pyglet.image.Texture3D class method), 243
create_for_images() (pyglet.image.Texture3D class method), 243
create_for_size() (pyglet.image.Texture class method), 239
create_image() (CheckerImagePattern method), 214
create_image() (ImagePattern method), 237
create_image() (SolidColorImagePattern method), 238
create_indexed_domain() (in module pyglet.graphics.vertexdomain), 183
create_mappable_buffer() (in module pyglet.graphics.vertexbuffer), 176
create_search_pattern() (FontConfig method), 121
create_texture() (ImageData method), 227
CURSOR_CROSSHAIR (Window attribute), 442
CURSOR_DEFAULT (Window attribute), 442
CURSOR_HAND (Window attribute), 442
CURSOR_HELP (Window attribute), 442
CURSOR_NO (Window attribute), 442
CURSOR_SIZE (Window attribute), 442
CURSOR_SIZE_DOWN (Window attribute), 442
CURSOR_SIZE_DOWN_LEFT (Window attribute), 442
CURSOR_SIZE_DOWN_RIGHT (Window attribute), 442
CURSOR_SIZE_LEFT (Window attribute), 442
CURSOR_SIZE_LEFT_RIGHT (Window attribute), 442
CURSOR_SIZE_RIGHT (Window attribute), 442
CURSOR_SIZE_UP (Window attribute), 442
CURSOR_SIZE_UP_DOWN (Window attribute), 442
CURSOR_SIZE_UP_LEFT (Window attribute), 442
CURSOR_SIZE_UP_RIGHT (Window attribute), 442
CURSOR_TEXT (Window attribute), 442
CURSOR_WAIT (Window attribute), 442
CURSOR_WAIT_ARROW (Window attribute), 442
D
data (ImageData attribute), 229
data (ImageDataRegion attribute), 231
dealloc() (Allocator method), 151
decode() (AttributedTextDecoder method), 364
decode() (DocumentDecoder method), 399
decode() (ImageDecoder method), 198
decode() (PlainTextDecoder method), 367
decode() (StructuredTextDecoder method), 371
decode_animation() (ImageDecoder method), 198
decode_attribute() (in module pyglet.texture), 414
decode_html() (in module pyglet.texture), 414
decode_structured() (HTMLDecoder method), 365
decode_structured() (StructuredTextDecoder method), 371
decode_text() (in module pyglet.texture), 414
decorate_function() (in module pyglet.gl.lib), 146
DEFAULT_MODE (in module pyglet.gl), 138
default_style (HTMLDecoder attribute), 365
DefaultMouseCursor (class in pyglet.window), 424
deinterlace() (Reader method), 110
delete() (AbstractAudioPlayer method), 293
delete() (AbstractBuffer method), 168
delete() (Caret method), 342
delete() (IncrementalTextLayout method), 376
delete() (IndexedVertedList method), 179
delete() (Player method), 303
delete() (RunList method), 398
delete() (SilentAudioDriver method), 269
delete() (SilentAudioPlayerPacketConsumer method), 270
delete() (SilentTimeAudioPlayer method), 271
delete() (Sprite method), 335
delete() (TextLayout method), 387
delete() (Texture method), 240
delete() (VertexArray method), 173
delete() (VertexBufferObject method), 175
delete() (VertedList method), 182
delete_text() (AbstractDocument method), 348
DepthBufferImage (class in pyglet.image), 220
DepthTexture (class in pyglet.image), 222
 Index 477
FcMatchPattern (in module pyglet.font.fontconfig), 125
FcResultMatch (in module pyglet.font.fontconfig), 125
FcResultNoId (in module pyglet.font.fontconfig), 125
FcResultNoMatch (in module pyglet.font.fontconfig), 126
FcResultOutOfMemory (in module pyglet.font.fontconfig), 126
FcResultTypeMismatch (in module pyglet.font.fontconfig), 126
FcTypeBool (in module pyglet.font.fontconfig), 126
FcTypeCharSet (in module pyglet.font.fontconfig), 126
FcTypeDouble (in module pyglet.font.fontconfig), 126
FcTypeFTFace (in module pyglet.font.fontconfig), 127
FcTypeInteger (in module pyglet.font.fontconfig), 127
FcTypeLangSet (in module pyglet.font.fontconfig), 127
FcTypeMatrix (in module pyglet.font.fontconfig), 127
FcTypeString (in module pyglet.font.fontconfig), 127
FcTypeVoid (in module pyglet.font.fontconfig), 128
FcValue (class in pyglet.font.fontconfig), 121
file (FontConfigSearchResult attribute), 123
file (in module pyglet.resource), 330
file() (Loader method), 325
file_scanlines() (Writer method), 113
FileLocation (class in pyglet.resource), 323
filter_scanline() (in module pyglet.extlibs.png), 116
FilteredRunIterator (class in pyglet.text.runlist), 395
find_font() (FontConfig method), 121
flip() (Window method), 434
fog_coords (VertexList attribute), 182
FogCoordAttribute (class in pyglet.graphics.vertexattribute), 157
font (Text attribute), 134
font_name (DocumentLabel attribute), 401
font_size (DocumentLabel attribute), 401
font_sizes (HTMLDecoder attribute), 366
FontConfig (class in pyglet.font.fontconfig), 121
FontConfigPattern (class in pyglet.font.fontconfig), 122
FontConfigSearchPattern (class in pyglet.font.fontconfig), 122
FontConfigSearchResult (class in pyglet.font.fontconfig), 123
foreground_decoration_group (TextLayout attribute), 389
foreground_group (TextLayout attribute), 389
format (BufferImage attribute), 208
format (BufferImageMask attribute), 211
format (ColorBufferImage attribute), 215
format (DepthBufferImage attribute), 220
format (ImageData attribute), 229
format_re (OrderedListBuilder attribute), 370
FormatError, 115
FormattedDocument (class in pyglet.text.document), 352
forward_orientation (AbstractListener attribute), 294
FPSDisplay (class in pyglet.window), 426
from_array() (in module pyglet.extlibs.png), 117
from_image_sequence() (pyglet.image.Animation class method), 206
fromarray() (in module pyglet.extlibs.png), 117
generic (Window attribute), 443
generators (in module pyglet.extlibs.png), 119
GenericAttribute (class in pyglet.graphics.vertexattribute), 159
genre (SourceInfo attribute), 312
get() (TextureGrid method), 248
get_allocated_regions() (Allocator method), 151
get_animation() (AbstractImageSequence method), 205
get_animation() (Source method), 308
get_animation_decoders() (in module pyglet.image.codecs), 200
get_animation_file_extensions() (ImageDecoder method), 198
get_apple_remote() (in module pyglet.input), 266
get_audio_data() (ProceduralSource method), 273
get_audio_data() (Source method), 308
get_audio_data() (SourceGroup method), 310
get_audio_data() (StaticMemorySource method), 313
get_audio_data() (StaticSource method), 315
get_audio_data() (WaveSource method), 288
get_audio_driver() (in module pyglet.media), 321
get_aux_buffer() (BufferManager method), 213
get_break_index() (GlyphString method), 132
get_buffer_manager() (in module pyglet.image), 262
get_buffer_mask() (BufferManager method), 213
get_cached_animation_names (in module pyglet.resource), 330
get_cached_animation_names() (Loader method), 325
get_cached_image_names (in module pyglet.resource), 331
get_cached_image_names() (Loader method), 325
get_cached_textures_names (in module pyglet.resource), 331
get_cached_textures_names() (Loader method), 325
generate() (TruetypeInfo method), 129
generate() (TrueTypeInfo method), 129
generate() (RiffForm method), 285
generate() (SourceGroup method), 310
data() (ImageData method), 227
data() (ImageDataRegion method), 231
data() (RIFFChunk method), 284
data() (WaveForm method), 287
data() (pyglet.texture) method, 415
data_re (pyglet.texture) method, 429
get_default() (in module pyglet.clock), 98
get_default_display() (Platform method), 429

478 Index
get_default_screen() (Display method), 426
get_depth_buffer() (BufferManager method), 214
get_devices() (in module pyglet.input), 267
get_display() (in module pyglet.canvas), 91
get_display() (Platform method), 429
get_domain() (VertexList method), 182
get_duration() (Animation method), 206
get_element() (AbstractDocument method), 348
get_element_runs() (FormattedDocument method), 353
get_element_runs() (UnformattedDocument method), 359
get_encoders() (in module pyglet.image.codecs), 200
get_extensions() (in module pyglet.gl.gl_info), 141
get_extensions() (GLInfo method), 139
get_extensions() (GLUInfo method), 143
get_file_extensions() (ImageDecoder method), 198
get_font() (AbstractDocument method), 248
get_font() (FormattedDocument method), 353
get_font() (UnformattedDocument method), 359
get_font_runs() (AbstractDocument method), 348
get_font_runs() (FormattedDocument method), 353
get_font_runs() (UnformattedDocument method), 359
get_font_selection_flags() (TruetypeInfo method), 129
get_fontconfig() (in module pyglet.font.fontconfig), 124
get_format_chunk() (WaveForm method), 287
get_fps() (Clock method), 94
get_fps() (in module pyglet.clock), 98
get_fps_limit() (Clock method), 94
get_fps_limit() (in module pyglet.clock), 99
get_fragmentation() (Allocator method), 151, 194
get_fracted_free_size() (Allocator method), 151
glyph advances() (TruetypeInfo method), 129
glyph kernings() (TruetypeInfo method), 129
glyph map() (TruetypeInfo method), 130
get_horizontal_metrics() (TruetypeInfo method), 130
get_image() (HTMLDecoder method), 365
get_image_data() (AbstractImage method), 202
get_image_data() (BufferImage method), 208
get_image_data() (ImageData method), 228
get_image_data() (ImageGrid method), 234
get_image_data() (Texture method), 240
get_image_data() (TextureRegion method), 252
get_index_region() (IndexedVertexDomain method), 178
get_job() (WorkerThread method), 319
get_joysticks() (in module pyglet.input), 267
get_line_count() (IncrementalTextLayout method), 376
get_line_count() (IncrementalTextLayout method), 376
get_line_from_point() (IncrementalTextLayout method), 376
get_location() (Window method), 434
get_mark() (ListBuilder method), 369
get_mark() (OrderedListBuilder method), 370
get_mark() (UnorderedListBuilder method), 372
get_max_height() (Animation method), 206
get_max_width() (Animation method), 206
get_mipmapped_texture() (AbstractImage method), 203
get_mipmapped_texture() (CompressedImageData method), 218
get_mipmapped_texture() (ImageData method), 228
get_name() (TruetypeInfo method), 130
get_names() (TruetypeInfo method), 130
get_next_video_frame() (Source method), 309
get_next_video_frame() (SourceGroup method), 310
get_next_video_timestamp() (Source method), 309
get_next_video_timestamp() (SourceGroup method), 311
get_paragraph_end() (AbstractDocument method), 349
get_paragraph_start() (AbstractDocument method), 349
get_platform() (in module pyglet.window), 446
get_point_from_line() (IncrementalTextLayout method), 376
get_point_from_position() (IncrementalTextLayout method), 376
get_position_from_line() (IncrementalTextLayout method), 377
get_position_from_point() (IncrementalTextLayout method), 377
get_position_on_line() (IncrementalTextLayout method), 377
get_region() (AbstractAttribute method), 154
get_region() (AbstractImage method), 203
get_region() (AbstractMappable method), 170
get_region() (BufferImage method), 208
get_region() (ImageData method), 228
get_region() (ImageRegion method), 231
get_region() (MappableVertexBufferObject method), 172
get_region() (Texture method), 240
get_region() (TextureRegion method), 252
get_region() (TileableTexture method), 257
get_region() (VertexArray method), 173
get_renderer() (in module pyglet.gl.gl_info), 141
get_renderer() (GLInfo method), 140
get_run_iterator() (RunList method), 398
gt_screens() (Display method), 426
get_script_home() (in module pyglet.resource), 329
get_settings_path() (in module pyglet.resource), 329
gt_silent_audio_driver() (in module pyglet.media), 321
get_size() (VertexBufferObject method), 182
get_size() (Window method), 434
gt_sleep_time() (Clock method), 94
get_sleep_time() (in module pyglet.clock), 99
get_source_loader() (in module pyglet.resource), 329
get_string_data() (AudioData method), 296
get_style() (AbstractDocument method), 349
get_style() (Caret method), 343
get_style() (DocumentLabel method), 401
get_style() (FormattedDocument method), 353
get_style() (UnformattedDocument method), 359
get_style_range() (AbstractDocument method), 349
get_style_runs() (AbstractDocument method), 349
get_style_runs() (FormattedDocument method), 353
get_style_runs() (UnformattedDocument method), 359
get_subwidth() (GlyphString method), 132
get_system_mouse_cursor() (Window method), 434
get_tablets() (in module pyglet.input), 267
get_texture() (AbstractImage method), 203
get_texture() (ColorBufferImage method), 215
get_texture() (CompressedImageData method), 218
get_texture() (DepthBufferImage method), 220
get_texture() (ImageData method), 228
get_texture() (ImageGrid method), 235
get_texture() (Player method), 303
get_texture() (Texture method), 240
get_texture_bins (in module pyglet.resource), 331
get_texture_bins() (Loader method), 325
get_texture_sequence() (AbstractImageSequence method), 205
get_texture_sequence() (ImageGrid method), 235
get_texture_sequence() (TextureSequence method), 255
get_time() (AbstractAudioPlayer method), 293
get_time() (SilentAudioPlayerPacketConsumer method), 270
get_time() (SilentTimeAudioPlayer method), 271
get_transform() (Animation method), 206
get_transform() (Texture method), 240
get_usage() (Allocator method), 151, 194
get_vendor() (GLInfo method), 140
get_version (in module pyglet.gl.gl_info), 141
get_version() (GLInfo method), 140
get_version() (GLUInfo method), 143
get_viewport() (BufferManager method), 214
get_wave_form() (RIFFFile method), 284
get_windows() (Display method), 426
GLException, 145
GLInfo (in module pyglet.gl.gl_info), 139
GLUInfo (class in pyglet.gl.gl_info), 142
GlyphString (class in pyglet.font), 131
Group (class in pyglet.graphics), 186
group (Sprite attribute), 336
group() (in module pyglet.extlibs.png), 118

H

halign (Text attribute), 134
handle_charref() (HTMLDecoder method), 365
handle_data() (HTMLDecoder method), 365
handle_endtag() (HTMLDecoder method), 365
handle_entityref() (HTMLDecoder method), 365
has_exit (Window attribute), 443
has_exit (WindowExitHandler attribute), 419
has_next() (SourceGroup method), 311
have_avbin (in module pyglet.media), 321
have_context (GLInfo attribute), 140
have_context (GLUInfo attribute), 144
have_context() (in module pyglet.gl.gl_info), 140
have_extension (in module pyglet.media), 321
have_extension (in module pyglet.gl.gl_info), 141
have_extension (in module pyglet.gl.gl_info), 144
have_extension() (GLInfo method), 140
have_extension() (GLUInfo method), 143
have_font() (in module pyglet.font), 136
have_version (in module pyglet.gl.gl_info), 141
have_version (in module pyglet.gl.gl_info), 144
have_version() (GLInfo method), 140
have_version() (GLUInfo method), 143
header_fmt (RIFFChunk attribute), 284
header_length (RIFFChunk attribute), 284
height (IncrementalTextLayout attribute), 378
height (ScrollBarTextLayout attribute), 382
height (ScrollBarTextLayoutGroup attribute), 385
height (Sprite attribute), 336
height (Text attribute), 134
height (TextStyle attribute), 289
height (Window attribute), 443
html (in module pyglet.resource), 331
html() (Loader method), 325
HTMLDecoder (class in pyglet.text.formats.html), 365
HTMLLabel (class in pyglet.text), 404
IBM_FORMAT_ADPCM (in module pyglet.media.riff), 290
IBM_FORMAT_ALAW (in module pyglet.media.riff), 290
IBM_FORMAT_MULAW (in module pyglet.media.riff), 291
Image (class in pyglet.extlibs.png), 107
image (in module pyglet.font), 136
image (in module pyglet.resource), 331
image (Sprite attribute), 336
image() (Loader method), 326
image_data (AbstractImage attribute), 204
image_data (Texture attribute), 240
ImageData (class in pyglet.image), 226
ImageDataRegion (class in pyglet.image), 230
ImageDecodeException, 199
ImageDecoder (class in pyglet.image.codecs), 198
ImageElement (class in pyglet.text.formats.structured), 368
ImageEncodeException, 200
ImageEncoder (class in pyglet.image.codecs), 199
move_to_point() (Caret method), 343
multi_tex_coords (VertexList attribute), 182
multiline (IncrementalTextLayout attribute), 378
multiline (TextLayout attribute), 389
MultiTexCoordAttribute (class in pyglet.graphics.vertexattribute), 160

N
name (FontConfigSearchResult attribute), 123
next() (ConstRunIterator method), 395
next() (Player method), 303
next() (RunIterator method), 397
next() (SourceGroup method), 311
next_source() (Player method), 303
null_group (in module pyglet.graphics), 191
null_group (in module pyglet.graphics.vertexattribute), 161
normals (VertexList attribute), 182
NoSuchConfigException, 445
NoSuchDisplayException, 446
NoSuchScreenModeException, 446
notify() (MediaThread method), 302
null_group (in module pyglet.graphics), 191
NullGroup (class in pyglet.graphics), 187

O
on_activate() (Caret method), 343
on_activate() (Window method), 437
on_activate() (WindowEventLogger method), 418
on_animation_end() (Sprite method), 335
on_close() (Window method), 437
on_close() (WindowEventLogger method), 418
on_context_lost() (Window method), 437
on_context_lost() (WindowEventLogger method), 418
on_context_state_lost() (Window method), 438
on_context_state_lost() (WindowEventLogger method), 418
on_deactivate() (Caret method), 343
on_deactivate() (Caret method), 343
on_deactivate() (Window method), 438
on_deactivate() (WindowEventLogger method), 418
on_delete_text() (AbstractDocument method), 350
on_delete_text() (IncrementalTextLayout method), 377
on_delete_text() (TextLayout method), 388
on_draw() (Window method), 438
on_draw() (WindowEventLogger method), 418
on_eos() (Player method), 304
on_text() (Caret method), 343
on_text() (Window method), 438
on_text() (WindowEventLogger method), 418
on_text() (WindowEventLogger method), 418
on_text() (WindowEventLogger method), 419
on_text() (WindowEventLogger method), 419
on_text() (WindowEventLogger method), 419
on_text() (WindowEventLogger method), 419
on_text() (WindowEventLogger method), 419
on_text() (WindowEventLogger method), 419
opacity (Sprite attribute), 336
open() (FileLocation method), 323
open() (Location method), 327
open() (URLLocation method), 328
open() (ZIPLocation method), 328
options (in module pyglet), 449
R
ranges() (AbstractRunIterator method), 394
ranges() (ConstRunIterator method), 395
ranges() (FilteredRunIterator method), 396
ranges() (OverriddenRunIterator method), 396
ranges() (RunIterator method), 397
ranges() (ZipRunIterator method), 399
read() (Reader method), 110
read_flat() (Reader method), 110
read_pam_header() (in module pyglet.extlibs.png), 119
read_pnm_header() (in module pyglet.extlibs.png), 119
Reader (class in pyglet.extlibs.png), 108
realloc() (Allocator method), 151
register_event_type() (pyglet.event.EventDispatcher class method), 104
reindex (in module pyglet.resource), 332
reindex() (Loader method), 326
remove() (ImageElement method), 368
remove() (InlineElement method), 358
remove() (WeakSet method), 89
remove_active_context (in module pyglet.gl.gl_info), 141
remove_active_context() (GLInfo method), 140
remove_handler() (EventDispatcher method), 104
remove_handlers() (EventDispatcher method), 104
renderer (GLInfo attribute), 140
resizable (Window attribute), 443
resize() (AbstractBuffer method), 169
resize() (IndexedVertexList method), 179
resize() (MappableVertexBufferObject method), 172
resize() (VertexArray method), 173
resize() (VertexBufferObject method), 175
resize() (VertexList method), 182
ResourceNotFoundException, 329
RIFFChunk (class in pyglet.media.riff), 283
RIFFFile (class in pyglet.media.riff), 284
RIFFForm (class in pyglet.media.riff), 285
RIFFFormatException, 290
RIFFSourceLoader (class in pyglet.media), 307
RIFFType (class in pyglet.media.riff), 285
RIGHT (in module pyglet.window.mouse), 424
RIGHT (Text attribute), 134
rotation (Sprite attribute), 336
rows (TextureGrid attribute), 248
run() (in module pyglet.app), 89
run() (MediaThread method), 302
run() (WorkerThread method), 319
RunIterator (class in pyglet.text.runlist), 397
RunList (class in pyglet.text.runlist), 397
S
safe() (AttributedTextDecoder method), 364
safe_node() (AttributedTextDecoder method), 364
save() (AbstractImage method), 203
save() (Image method), 107
Saw (class in pyglet.media.procedural), 274
scale (Sprite attribute), 336
schedule() (Clock method), 95
schedule() (in module pyglet.clock), 99
schedule_interval() (Clock method), 95
schedule_interval() (in module pyglet.clock), 99
schedule_interval_soft() (Clock method), 95
schedule_interval_soft() (in module pyglet.clock), 99
schedule_once() (Clock method), 96
schedule_once() (in module pyglet.clock), 100
screen (Window attribute), 443
SCROLL_INCREMENT (Caret attribute), 344
ScrollableTextLayout (class in pyglet.text.layout), 381
ScrollableTextLayoutGroup (class in pyglet.text.layout), 384
secondary_colors (VertexList attribute), 182
SecondaryColorAttribute (class in pyglet.graphics.vertexattribute), 162
seek() (Player method), 303
seek() (ProceduralSource method), 273
seek() (Source method), 309
seek() (SourceGroup method), 311
seek() (StaticMemorySource method), 313
seek() (WaveSource method), 289
seek_next_frame() (Player method), 303
select_paragraph() (Caret method), 344
select_to_point() (Caret method), 344
select_word() (Caret method), 344
selection_background_color (IncrementalTextLayout attribute), 378
selection_color (IncrementalTextLayout attribute), 378
selection_end (IncrementalTextLayout attribute), 378
selection_start (IncrementalTextLayout attribute), 378
serialize_attributes() (in module pyglet.graphics.vertexattribute), 167
serialtoflat() (Reader method), 111
set_active_context (in module pyglet.gl.gl_info), 141
set_active_context (in module pyglet.gl.glu_info), 144
set_active_context() (GLInfo method), 140
set_active_context() (GLUInfo method), 143
set_capacity() (Allocator method), 151
set_caption() (Window method), 434
set_cone_inner_angle() (AbstractAudioPlayer method), 294
set_cone_orientation() (AbstractAudioPlayer method), 294
set_cone_outer_angle() (AbstractAudioPlayer method), 294
set_cone_outer_gain() (AbstractAudioPlayer method), 294
set_data() (AbstractBuffer method), 169
set_data() (ImageData method), 228
set_data() (MappableVertexBufferObject method), 172
set_data() (VertexArray method), 173
set_data() (VertexBufferObject method), 175
set_data_region() (AbstractBuffer method), 169
set_data_region() (MappableVertexBufferObject method), 172
set_data_region() (VertexArray method), 173
set_default() (in module pyglet.clock), 100
set_exclusive_keyboard() (Window method), 435
set_exclusive_mouse() (Window method), 435
set_fps() (FPSDisplay method), 427
set_fps_limit() (Clock method), 96
set_fps_limit() (in module pyglet.clock), 100
set_handler() (EventDispatcher method), 104
set_handlers() (EventDispatcher method), 105
set_icon() (Window method), 435
set_location() (Window method), 435
set_max_distance() (AbstractAudioPlayer method), 294
set_maximum_size() (Window method), 436
set_min_distance() (AbstractAudioPlayer method), 294
set_minimum_size() (Window method), 436
set_mipmap_data() (CompressedImageData method), 218
set_mipmap_image() (ImageData method), 228
set_mouse_cursor() (Window method), 436
set_mouse_platform_visible() (Window method), 436
set_mouse_visible() (Window method), 436
set_paragraph_style() (AbstractDocument method), 350
set_paragraph_style() (UnformattedDocument method), 359
set_pitch() (AbstractAudioPlayer method), 294
set_pointer() (AbstractAttribute method), 154
set_pointer() (ColorAttribute method), 155
set_pointer() (EdgeFlagAttribute method), 157
set_pointer() (FogCoordAttribute method), 158
set_pointer() (GenericAttribute method), 159
set_pointer() (MultiTexCoordAttribute method), 160
set_pointer() (NormalAttribute method), 161
set_pointer() (SecondaryColorAttribute method), 163
set_pointer() (TexCoordAttribute method), 165
set_position() (AbstractAudioPlayer method), 294
set_position() (Sprite method), 335
set_region() (AbstractAttribute method), 154
set_run() (RunList method), 398
set_selection() (IncrementalTextLayout method), 377
set_size() (Window method), 436
set_state() (Group method), 186
set_state() (SpriteGroup method), 339
set_state() (TextLayoutGroup method), 392
set_state() (TextLayoutTextureGroup method), 392
set_state() (TextureGroup method), 189
set_state_recursive() (Group method), 186
set_style() (AbstractDocument method), 350
set_style() (Caret method), 344
set_style() (DocumentLabel method), 401
set_style() (UnformattedDocument method), 359
set_visible() (Window method), 437
set_volume() (AbstractAudioPlayer method), 294
set_vsync() (Window method), 437
Silence (class in pyglet.media.procedural), 276
SilentAudioDriver (class in pyglet.media.drivers.silent), 269
SilentAudioPacket (class in pyglet.media.drivers.silent), 269
SilentAudioPlayerPacketConsumer (class in pyglet.media.drivers.silent), 270
SilentTimeAudioPlayer (class in pyglet.media.drivers.silent), 271
Sine (class in pyglet.media.procedural), 278
size (AbstractBuffer attribute), 169
size (FontConfigSearchResult attribute), 123
sleep() (MediaThread method), 302
SLEEP_UNDERSHOOT (Clock attribute), 97
SolidColorImagePattern (class in pyglet.image), 237
Source (class in pyglet.media), 308
source (Player attribute), 305
SourceGroup (class in pyglet.media), 310
SourceInfo (class in pyglet.media), 311
Sprite (class in pyglet.sprite), 334
SpriteGroup (class in pyglet.sprite), 339
Square (class in pyglet.media.procedural), 280
start() (MediaThread method), 302
StaticMemorySource (class in pyglet.media), 312
StaticSource (class in pyglet.media), 314
stop() (AbstractAudioPlayer method), 294
stop() (MediaThread method), 302
stop() (SilentAudioPlayerPacketConsumer method), 270
stop() (SilentTimeAudioPlayer method), 271
StreamingSource (class in pyglet.media), 316
StructuredTextDecoder (class in pyglet.text.formats.structured), 371
style (Window attribute), 443
STYLE_INDETERMINATE (in module pyglet.text.document), 363
symbol_string() (in module pyglet.window.key), 423
T
test_clock() (in module pyglet.clock), 100
tests.test (module), 464
tex_coords (Texture attribute), 240
tex_coords (VertexList attribute), 182
tex_coords_order (Texture attribute), 241
TexCoordAttribute (class in pyglet.graphics.vertexattribute), 164
text (AbstractDocument attribute), 350
Text (class in pyglet.font), 133
document (DocumentLabel attribute), 402
text (HTMLLabel attribute), 405
text (in module pyglet.resource), 332
text (Text attribute), 135
text (Loader method), 326
TextLayout (class in pyglet.text.layout), 386
TextLayoutForegroundDecorationGroup (class in pyglet.text.layout), 389
TextLayoutForegroundGroup (class in pyglet.text.layout), 390
TextLayoutGroup (class in pyglet.text.layout), 391
TextLayoutTextureGroup (class in pyglet.text.layout), 392
texture (AbstractImage attribute), 204
texture (class in pyglet.image), 238
texture (in module pyglet.resource), 332
texture (Loader method), 326
Texture (class in pyglet.image), 238
TextureAtlas (class in pyglet.image.sequence), 205
TextureBin (class in pyglet.image.atlas), 195
TextureGrid (class in pyglet.image), 246
TextureGroup (class in pyglet.graphics), 189
TextureRegion (class in pyglet.image), 251
TextureSequence (class in pyglet.image), 255
tick() (Clock method), 96
tick() (in module pyglet.clock), 100
TileableTexture (class in pyglet.image), 256
time (Player attribute), 305
title (SourceInfo attribute), 312
top (ScrollableTextLayoutGroup attribute), 385
TOP (Text attribute), 134
top_group (TextLayout attribute), 389
tostring() (in module pyglet.extlibs.png), 119
track (SourceInfo attribute), 312
translate_timestamp() (SourceGroup method), 311
translate_x (ScrollableTextLayoutGroup attribute), 385
translate_y (ScrollableTextLayoutGroup attribute), 385
TruetypeInfo (class in pyglet.font.ttf), 128
type (FcValue attribute), 121

V
validate_signature() (Reader method), 111
valign (Text attribute), 135
vendor (GLInfo attribute), 140
version (GLInfo attribute), 140
version (GLUInfo attribute), 144
version (in module pyglet), 450
vertex_list() (in module pyglet.graphics), 190
vertex_list_indexed() (in module pyglet.graphics), 191
VertexArray (class in pyglet.graphics.vertexbuffer), 173
VertexArrayRegion (class in pyglet.graphics.vertexbuffer), 174
VertexAttribute (class in pyglet.graphics.vertexattribute), 165
VertexBufferObject (class in pyglet.graphics.vertexbuffer), 174
VertexBufferObjectRegion (class in pyglet.graphics.vertexbuffer), 176
VertexBufferObject (class in pyglet.graphics.vertexbuffer), 174
VertexAttribute (class in pyglet.graphics.vertexattribute), 165
vertices (VertexList attribute), 183
video_format (Source attribute), 309
VideoFormat (class in pyglet.media), 318
view_x (ScrollableTextLayout attribute), 382
view_x (ScrollableTextLayoutGroup attribute), 385
view_y (IncrementalTextLayout attribute), 378
view_y (ScrollableTextLayout attribute), 382
view_y (ScrollableTextLayoutGroup attribute), 385
visible (Caret attribute), 345

U
u (FcValue attribute), 121
unbind() (AbstractBuffer method), 169
unbind() (VertexArray method), 173
unbind() (VertexBufferObject method), 175
undo_filter() (Reader method), 111
UnformattedDocument (class in pyglet.text.document), 358
UniformTextureSequence (class in pyglet.image), 260
unbind() (MappableVertexBufferObject method), 172
unbind() (VertexArray method), 174
unbind() (VertexBufferObject method), 175
UnorderedListBuilder (class in pyglet.text.formats.structured), 372
unschedule() (Clock method), 96
unschedule() (ClockDisplay method), 98
unschedule() (in module pyglet.clock), 101
unset_state() (Group method), 186
unset_state() (ScrollableTextLayoutGroup method), 385
unset_state() (SpriteGroup method), 339
unset_state() (TextLayoutGroup method), 392
unset_state() (TextureGroup method), 189
unset_state_recursive() (Group method), 187
up_orientation (AbstractListener attribute), 295
update() (FPSDisplay method), 427
update_period (FPSDisplay attribute), 427
update_text() (ClockDisplay method), 98
update_texture() (Player method), 304
update_time() (Clock method), 96
URLLocation (class in pyglet.resource), 327
user_key() (in module pyglet.window.key), 423

"
visible (Sprite attribute), 336
visible (Window attribute), 443
volume (AbstractListener attribute), 295
volume (Player attribute), 305
vsync (Window attribute), 443

W
WAVE_FORMAT_PCM (in module pyglet.media.riff), 291
WaveDataChunk (class in pyglet.media.riff), 286
WaveForm (class in pyglet.media.riff), 287
WaveFormatChunk (class in pyglet.media.riff), 287
WAVEFormatException, 290
WaveSource (class in pyglet.media.riff), 288
WeakSet (class in pyglet.app), 88
WhiteNoise (class in pyglet.media.procedural), 281
width (IncrementalTextLayout attribute), 378
width (ScrollableTextLayout attribute), 382
width (ScrollableTextLayoutGroup attribute), 385
width (Sprite attribute), 336
width (Text attribute), 135
width (TextLayout attribute), 389
width (Window attribute), 443
Window (class in pyglet.window), 430
window (in module pyglet.font), 136
WINDOW_STYLE_BORDERLESS (Window attribute), 442
WINDOW_STYLE_DEFAULT (Window attribute), 442
WINDOW_STYLE_DIALOG (Window attribute), 442
WINDOW_STYLE_TOOL (Window attribute), 442
WindowEventLogger (class in pyglet.window.event), 417
WindowException, 446
WindowExitHandler (class in pyglet.window.event), 419
windows (in module pyglet.app), 90
WorkerThread (class in pyglet.media), 319
write() (Writer method), 113
write_array() (Writer method), 113
write_chunk() (in module pyglet.extlibs.png), 119
write_chunks() (in module pyglet.extlibs.png), 119
write_packed() (Writer method), 114
write_passes() (Writer method), 114
write_pnm() (in module pyglet.extlibs.png), 119
Writer (class in pyglet.extlibs.png), 111

X
x (ScrollableTextLayout attribute), 382
x (Sprite attribute), 337
x (Text attribute), 135
x (TextLayout attribute), 389
x (Texture attribute), 241

Y
y (ScrollableTextLayout attribute), 382
y (Sprite attribute), 337
y (Text attribute), 135
y (TextLayout attribute), 389
y (Texture attribute), 241
year (SourceInfo attribute), 312

Z
z (Text attribute), 135
z (Texture attribute), 241
ZipLocation (class in pyglet.resource), 328
ZipRunIterator (class in pyglet.text.runlist), 398