pyglet Programming Guide

pyglet Programming Guide

Table of Contents

Welcome

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*.

Sections

- Installation
- Writing a pyglet application
- Creating an OpenGL context
- The OpenGL interface
- Graphics
- Windowing
- The application event loop
- The pyglet event framework
- Working with the keyboard
- Working with the mouse
- Keeping track of time
- Displaying text
- Images
- Sound and video
- Application resources
- Debugging tools
- Appendix: Migrating to pyglet 1.1

Table of contents

Installation

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

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

On Windows you can specify this from a command line:

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

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

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

or, using tcsh or a variant:

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

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

Installing using setup.py

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 priveleged 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 priveleged user; for example:

```
sudo easy_install -U pyglet
```

Writing a pyglet application

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

Hello, World

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 *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 alignment 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,
                          halign='center', valign='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 *pyglet.resource.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 *AbstractImage.blit* method draws the image. The arguments `(0, 0)` tell pyglet to draw the image at pixel coordinates 0, 0 in the window (the lower-left corner).

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

Handling mouse and keyboard events

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

```
import pyglet

window = pyglet.window.Window()

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

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

pyglet.app.run()
```

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

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

```
from pyglet.window import key

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

```
        print 'The enter key was pressed.'
```

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

Mouse events are handled in a similar way:

```
from pyglet.window import mouse

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

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

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

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

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

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

Playing sounds and music

pyglet makes it easy to play and mix multiple sounds together in your game. The following example plays an MP3 file [5]:

```
import pyglet

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

pyglet.app.run()
```

As with the image loading example presented earlier, *pyglet.resource.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 *pyglet.media.load*.

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

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

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

[5] 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.

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.

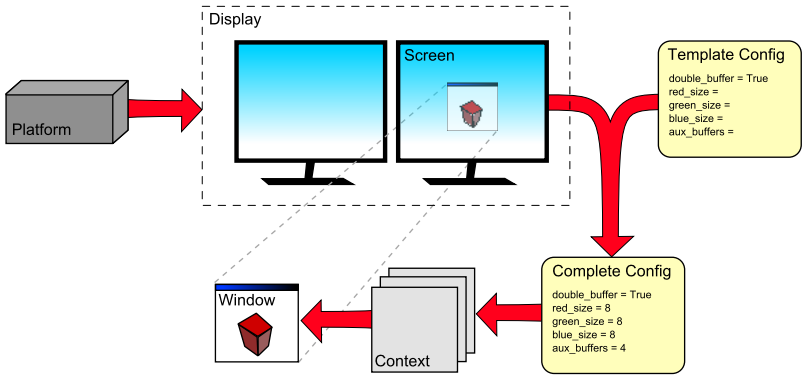
To write 3D applications or achieve optimal performance in your 2D applications you'll need to work with OpenGL directly. The canonical references for OpenGL are The OpenGL Programming Guide [http://opengl.org/documentation/books/#the_opengl_programming_guide_the_official_guide_to_learning_opengl_version] and The OpenGL Shading Language [http://opengl.org/documentation/books/#the_opengl_shading_language_2nd_edition].

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

Creating an OpenGL context

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

Displays, screens, configs and contexts



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

Contexts and configs

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

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

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

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

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

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

Displays

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

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

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

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

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

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

You use the same string to specify a separate X11 screen [6]:

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

Screens

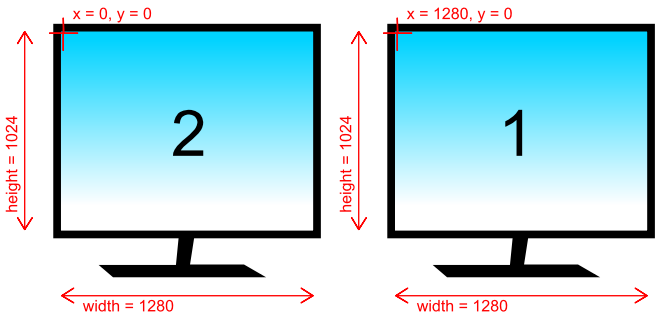
Once you have obtained a display, you can enumerate the screens that are connected. A screen is the physical display medium connected to the display device; for example a computer monitor, TV or projector. Most computers will have a single screen, however dual-head workstations and laptops connected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

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

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

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

[6] 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.



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

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

OpenGL configuration options

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

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

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

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

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

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

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

`sample_buffers and samples`

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

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

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

`stereo`

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

`double_buffer`

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

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

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

`depth_size`

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

`stencil_size`

The stencil buffer is required for masking the other buffers and implementing certain volumetric shadowing algorithms. The typical

	stencil size is 8 bits; or specify `0` if you do not require it.
`accum_red_size`, `accum_blue_size`, `accum_green_size`, `accum_alpha_size`	The accumulation buffer can be used for simple antialiasing, depth-of-field, motion blur and other compositing operations. Its use nowadays is being superceded by the use of floating-point textures, however it is still a practical solution for implementing these effects on older hardware. If you require an accumulation buffer, specify `8` for each of these attributes (the alpha component is optional, of course).
`aux_buffers`	Each auxilliary buffer is configured the same as the colour buffer. Up to four auxilliary buffers can typically be created. Specify `0` if you do not require any auxilliary buffers. Like the accumulation buffer, auxilliary buffers are used less often nowadays as more efficient techniques such as render-to-texture are available. They are almost universally available on older hardware, though, where the newer techniques are not possible.

The default configuration

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

Attribute	**Value**
double_buffer	True
depth_size	24

Simple context configuration

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

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

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

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

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

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

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

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

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

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

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

Selecting the best configuration

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

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

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

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

for config in screen.get_matching_configs(gl.Config()):
    if config.aux_buffers or config.accum_red_size:
        print config
```

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

Sharing objects between contexts

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

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

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

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

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

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

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

pyglet creates a hidden "shadow" context as soon as *pyglet.gl* is imported. By default, all windows will share object space with this shadow context, so the above steps are generally not needed. The shadow context also allows objects such as textures to be loaded before a window is created.

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

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

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

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

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

The OpenGL interface

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

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

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

Using OpenGL

Documentation of OpenGL and GLU are provided at the OpenGL website [http://www.opengl.org] and (more comprehensively) in the OpenGL Programming Guide [http://opengl.org/documentation/red_book/].

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

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

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

```
from pyglet.gl import *

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

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

pyglet.app.run()
```

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

```
from pyglet.gl import *

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

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

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

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

pyglet.app.run()
```

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

Resizing the window

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

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

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

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

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

Error checking

By default, pyglet calls `glGetError` after every GL function call (except where such a check would be invalid). If an error is reported, pyglet raises `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`:

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

from pyglet.gl import *
```

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

Using extension functions

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

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

You can also easily check the version of OpenGL:

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

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

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

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

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

Using multiple windows

pyglet allows you to create and display any number of windows simultaneously. Each will be created with its own OpenGL context, however all contexts will share the same texture objects, display lists, shader programs, and so on, by default [7]. 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*.

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

AGL, GLX and WGL

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

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

```
if sys.platform == 'linux2':
    from pyglet.gl.glx import *
    glxCreatePbuffer(...)
elif sys.platform == 'darwin':
    from pyglet.gl.agl import *
    aglCreatePbuffer(...)
```

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`.

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

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 [http://opengl.org/documentation/red_book/] for a description of each of mode.

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

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

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

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

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

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

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

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

```
pyglet.graphics.draw_indexed(2, pyglet.gl.GL_POINTS,
    [0, 1, 2, 3],
    ('v2i', (10, 15, 30, 35))
)
```

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

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

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

Vertex attributes

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

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

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

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

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

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

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

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

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

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

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

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

Vertex lists

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

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

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

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

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

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

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

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

Updating vertex data

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

Vertex attribute	**Object attribute**
Vertex position	`vertices`
Color	`colors`
Edge flag	`edge_flags`
Fog coordinate	`fog_coords`
Normal	`normals`
Secondary color	`secondary_colors`
Texture coordinate	`tex_coords`
Generic attribute	*Inaccessible*

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

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

The attributes can also be selectively updated in-place:

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

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

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

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

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

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

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

Data usage

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

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

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

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

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

Indexed vertex lists

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

Batched rendering

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

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

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

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

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

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

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

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

```
batch.draw()
```

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

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

Setting the OpenGL state

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

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

before drawing vertex lists, and then

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

afterwards to avoid interfering with later drawing commands.

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

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

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

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

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

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

Hierarchical state

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

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

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

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

    def unset_state(self):
        glDisable(GL_TEXTURE_2D)

texture_enable_group = TextureEnableGroup()

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

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

    # No unset_state method required.

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

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

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

Sorting vertex lists

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

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

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

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

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

Batches and groups in other modules

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

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

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

batch.draw()
```

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

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

Windowing

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

Creating a window

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

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

The default parameters used are:

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

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

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

Context configuration

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

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

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

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

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

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

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

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

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

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

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

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

Fullscreen windows

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

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

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

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

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

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

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

Size and position

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

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

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

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

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

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

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

```
    print 'The window was resized to %dx%d' % (width, height)
```

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

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

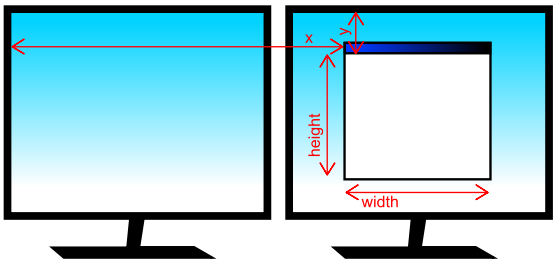
The window can also be resized programatically (even if the window is not user-resizable) with the *set_size* method:

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

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

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

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

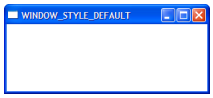
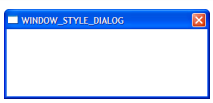
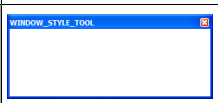


The position ond size of the window relative to the desktop.

Appearance

Window style

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

Style	Windows XP	Mac OS X
WINDOW_STYLE_DEFAULT		
WINDOW_STYLE_DIALOG		
WINDOW_STYLE_TOOL		

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

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

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

Caption

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

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

Icon

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

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

`Windows XP`	• A 16x16 icon for the title bar and task bar. • A 32x32 icon for the Alt+Tab switcher.
`Mac OS X`	• Any number of icons of resolutions 16x16, 24x24, 32x32, 48x48, 72x72 and 128x128. The actual image displayed will be interpolated to the correct size from those provided.
`Linux`	• No constraints, however most window managers will use a 16x16 and a 32x32 icon in the same way as Windows XP.

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

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

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

```
window.set_icon(icon1, icon2)
```

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

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

Visibility

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

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

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

Subclassing Window

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

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

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

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

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

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

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

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

Windows and OpenGL contexts

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

Double-buffering

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

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

Vertical retrace synchronisation

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

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

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

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

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

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

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

The application event loop

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

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

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

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

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

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

Customising the event loop

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

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

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

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

Event loop events

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

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

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

event_loop.run()
```

Overriding the default idle policy

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

The default implementation has the following algorithm:

1. Call *clock.tick* with `poll=True` to call any scheduled functions.

2. Dispatch the *on_draw* event and call *flip* on every open window.

3. Return the value of *clock.get_sleep_time*.

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

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

Dispatching events manually

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

A simple event loop usually has the following form:

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

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

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

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

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

- The *EventLoop* class provides plenty of hooks for most toolkits to be integrated without needing to resort to a manual event loop.

- Because *EventLoop* is tuned for specific operating systems, it is more responsive to user events, and continues calling clock functions while windows are being resized, and (on Mac OS X) the menu bar is being tracked.

- It is difficult to write a manual event loop that does not consume 100% CPU while still remaining responsive to user input.

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

The pyglet event framework

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

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

Setting event handlers

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

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

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

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

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

Note that you need not even name your function the same as the event.

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

```
window = window.Window()

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

You can even give the event handler another name if necessary:

```
@window.event('on_resize')
def handle_resize_event(width, height):
    pass
```

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

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

In this case you must use the name of the event handler as the method name.

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 propogate from the top of the stack to the bottom, but can be stopped by any handler along the way.

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

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

window.push_handlers(on_key_press)
```

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

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

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

window.push_handlers(on_key_press)
```

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

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

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

    window.push_handlers(on_key_press, on_mouse_press)

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

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

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

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

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

game_handlers = GameEventHandler()

def start_game()
    window.push_handlers(game_handlers)

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

Creating your own event dispatcher

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

The steps for creating an event dispatcher are:

1. Subclass *EventDispatcher*

2. Call the *register_event_type* class method on your subclass for each event your subclass will recognise.

3. Call *dispatch_event* to create and dispatch an event as needed.

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

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

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

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

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

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

```
widget = ClankingWidget()

@widget.event
def on_clank():
    pass

@widget.event
def on_clicked(clicks):
```

```
    pass

def override_on_clicked(clicks):
    pass

widget.push_handlers(on_clicked=override_on_clicked)
```

The *EventDispatcher* takes care of propogating 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 [Gamma,etal.,`DesignPatterns`Addison-Wesley1994], 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:

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

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

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

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

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

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

Documenting events

pyglet uses a modified version of Epydoc [http://epydoc.sourceforge.net/] to construct its API documentation. One of these modifications is the inclusion of an "Events" summary for event dispatchers. If you plan on releasing your code as a library for others to use, you may want to consider using the same tool to document code.

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

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

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

```
import sys

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

            :event:
            '''
```

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

Working with the keyboard

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

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

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

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

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

Keyboard events

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

Both events are parameterised by the same arguments:

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

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

Defined key symbols

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

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

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

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

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

Various control and directional keys are identified by name:

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

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

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

Keys on the number pad have separate symbols:

```
key.NUM_1
key.NUM_2
...
key.NUM_EQUAL
key.NUM_DIVIDE
key.NUM_MULTIPLY
key.NUM_MINUS
key.NUM_PLUS
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):

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

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

Modifiers

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

```
MOD_SHIFT
MOD_CTRL
```

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

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

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

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

User-defined key symbols

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

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

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

Remembering key state

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

```
from pyglet.window import key

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

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

Text and motion events

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

- The complex task of mapping modifiers and key symbols to Unicode characters is taken care of automatically and correctly.

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

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

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

```
def on_text(text):
    pass
```

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

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

Motion events

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

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

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

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

```
def on_text_motion(motion):
    pass
```

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

Constant	**Behaviour**	**Windows/ Linux**	**Mac OS X**
`MOTION_UP`	Move the cursor up	Up	Up
`MOTION_DOWN`	Move the cursor down	Down	Down

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

Keyboard exclusivity

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

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

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

The following restrictions apply on Windows:

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

The following restrictions apply on Mac OS X:

- The power key is not disabled.

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

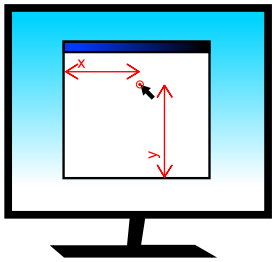
Working with the mouse

All pyglet windows can recieve 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 coordinate space for the mouse pointer.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
from pyglet.window import mouse

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

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

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

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

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

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

The mouse scroll wheel generates the *on_mouse_scroll* event:

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

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

Changing the mouse cursor

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

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

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

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

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

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

Constant	Windows XP	Mac OS X
CURSOR_DEFAULT		
CURSOR_CROSSHAIR		
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		
CURSOR_SIZE_RIGHT		
CURSOR_SIZE_UP		
CURSOR_SIZE_UP_DOWN		
CURSOR_SIZE_UP_LEFT		
CURSOR_SIZE_UP_RIGHT		
CURSOR_TEXT		
CURSOR_WAIT		
CURSOR_WAIT_ARROW		

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

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

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

Mouse exclusivity

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

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

Activate mouse exclusive mode with *set_exclusive_mouse*:

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

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

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

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

pyglet applications begin execution with:

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

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

Typical applications need to execute code in only three circumstances:

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

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

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

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

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

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

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

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

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

For one-shot events, use *schedule_once*:

```
def dismiss_dialog(dt):
```

```
    # ...

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

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

Animation techniques

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

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

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

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

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

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

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

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

The frame rate

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

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

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

Displaying the frame rate

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

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

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

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

User-defined clocks

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

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

Displaying text

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

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:

```
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()
```

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 `halign` and `valign` parameters.
- The actual text is drawn with the *Label.draw* method. Labels can also be added to a graphics batch; see *Graphics* for details.

Loading system fonts

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

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

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

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

```
                                        bold=True, italic=True)
```

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

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

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

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

Font sizes

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

Font resolution

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

The DPI can be specified directly in the *pyglet.font.load* function.

Determining font size

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

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

These measurements are shown in the diagram below.

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

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

```
ascent - descent + leading
```

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

Loading custom fonts

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

Loading a custom font must be performed in two steps:

1. Let pyglet know about the additional font or font files.

2. Load the font by its family name.

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

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

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

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

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

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

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

Supported font formats

pyglet can load any font file that the operating system natively supports. The list of supported formats is shown in the table below.

Font Format	**Windows XP**	**Mac OS X**	**Linux (FreeType)**
TrueType (.ttf)	X	X	X
PostScript Type 1 (.pfm, .pfb)	X	X	X
Windows Bitmap (.fnt)	X		X
Mac OS X Data Fork Font (.dfont)		X	
OpenType (.ttf) [8]		X	
X11 font formats PCF, BDF, SFONT			X
Bitstream PFR (.pfr)			X

[8] 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.

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 (for example, when reading the *width* or *height* properties of *Text*).

Blend state

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

The following blend state is used for drawing font glyphs:

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

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

Images

pyglet provides functions for loading and saving images in various formats using native operating system services. pyglet can also work with the Python Imaging Library [http://www.pythonware.com/products/pil/] (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:

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

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

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

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

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

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

pyglet provides the following image decoders:

Module	Class	Description
`pyglet.image.codecs.dds`	`DDSImageDecoder`	Reads Microsoft DirectDraw Surface files containing compressed textures
`pyglet.image.codecs.gdiplus`	`GDIPlusDecoder`	Uses Windows GDI+ services to decode images.
`pyglet.image.codecs.gdkpixbuf2`	`GdkPixbuf2ImageDecoder`	Uses the GTK-2.0 GDK functions to decode images.
`pyglet.image.codecs.pil`	`PILImageDecoder`	

Module	Class	Description
		Wrapper interface around PIL Image class.
pyglet.image.codecs.png	PNGImageDecoder	PNG decoder written in pure Python.
pyglet.image.codecs.quicktime	QuickTimeImageDecoder	Uses Mac OS X QuickTime to decode images.

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

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

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

This use is not recommended unless your application has to work around specific deficiences 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 [http://www.pythonware.com/library/pil/handbook/index.htm] for a list of such formats.

Extension	Description	Windows XP	Mac OS X	Linux [9]
.bmp	Windows Bitmap	X	X	X
.dds	Microsoft DirectDraw Surface [10]	X	X	X
.exif	Exif	X		
.gif	Graphics Interchange Format	X	X	X
.jpg .jpeg	JPEG/JIFF Image	X	X	X
.jp2 .jpx	JPEG 2000		X	
.pcx	PC Paintbrush Bitmap Graphic		X	

Extension	Description	Windows XP	Mac OS X	Linux [9]
`.png`	Portable Network Graphic	X	X	X
`.pnm`	PBM Portable Any Map Graphic Bitmap			X
`.ras`	Sun raster graphic			X
`.tga`	Truevision Targa Graphic		X	
`.tif .tiff`	Tagged Image File Format	X	X	X
`.xbm`	X11 bitmap		X	X
`.xpm`	X11 icon		X	X

[9]Requires GTK 2.0 or later.

[10]Only S3TC compressed surfaces are supported. Depth, volume and cube textures are not supported.

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

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

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

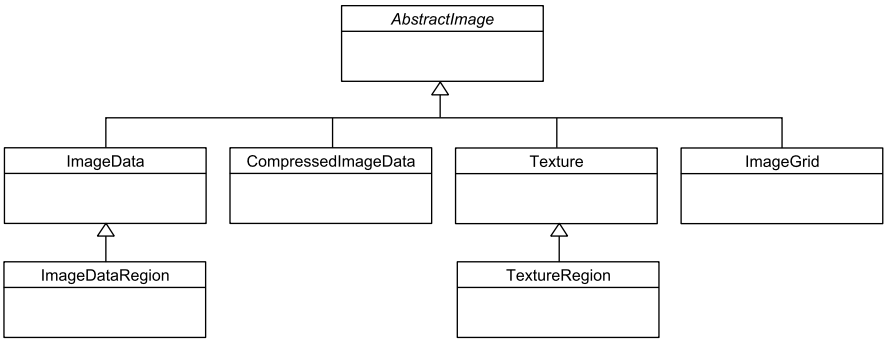
```
image_part = kitten.get_region(x=10, y=10, width=100, height=100)
```

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

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

The AbstractImage hierarchy

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



The *AbstractImage* class hierarchy.

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

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

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

Original class	**`.get_texture()`**	**`.get_image_data()`**
Texture	No change	`glGetTexImage2D`
TextureRegion	No change	`glGetTexImage2D`, crop resulting image.
ImageData	`glTexImage2D` [1]	No change
ImageDataRegion	`glTexImage2D` [1]	No change
CompressedImageData	`glCompressedTexImage2D` [2]	N/A [3]
BufferImage	`glCopyTexSubImage2D` [4]	`glReadPixels`

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

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

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

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

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

Accessing or providing pixel data

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

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

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

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

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

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

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

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

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

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

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

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

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

Performance concerns

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

1. Can the data be provided directly using a built-in OpenGL pixel format such as `GL_RGB` or `GL_RGBA`?

2. Is there an extension present that handles this pixel format?

3. Can the data be transformed with a single regular expression?

4. If none of the above are possible, the image will be split into separate scanlines and a regular expression replacement done on each; then the lines will be joined together again.

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

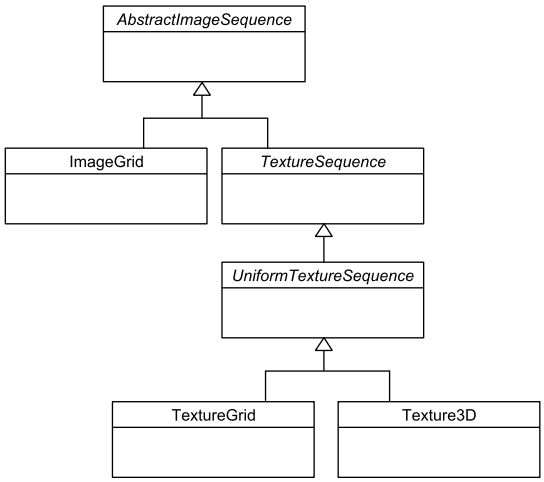
Format	**Required extensions**
`"I"`	
`"L"`	
`"LA"`	
`"R"`	
`"G"`	
`"B"`	
`"A"`	
`"RGB"`	
`"RGBA"`	
`"ARGB"`	`GL_EXT_bgra` and `GL_APPLE_packed_pixels`
`"ABGR"`	`GL_EXT_abgr`
`"BGR"`	`GL_EXT_bgra`
`"BGRA"`	`GL_EXT_bgra`

If the image data is not in one of these formats, a regular expression will be constructed to pull it into one. If the rows are not tightly packed, or if the image is ordered from top-to-bottom, the rows will be split before the regular expression is applied. Each of these may incur a performance penalty -- you should avoid such formats for real-time texture updates if possible.

Image sequences and atlases

Sometimes a single image is used to hold several images. For example, a "sprite sheet" is an image that contains each animation frame required for a character sprite animation.

pyglet provides convenience classes for extracting the individual images from such a composite image as if it were a simple Python sequence. Discrete images can also be packed into one or more larger textures with texture bins and atlases.



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.

An image consisting of eight animation frames arranged in a grid.

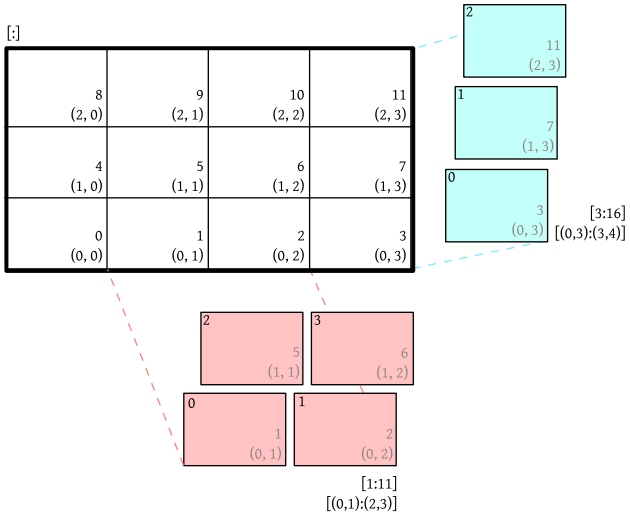
This image has one row and eight columns. This is all the information you need to create an *ImageGrid* with:

```
explosion = pyglet.image.load('explosion.png')
explosion_seq = pyglet.image.ImageGrid(explosion, 1, 8)
```

The images within the grid can now be accessed as if they were their own images:

```
frame_1 = explosion_seq[0]
frame_2 = explosion_seq[1]
```

Images with more than one row can be accessed either as a single-dimensional sequence, or as a (row, column) tuple; as shown in the following diagram.



An image grid with several rows and columns, and the slices that can be used to access it.

Image sequences can be sliced like any other sequence in Python. For example, the following obtains the first four frames in the animation:

```
start_frames = explosion_seq[:4]
```

For efficient rendering, you should use a *TextureGrid*. This uses a single texture for the grid, and each individual image returned from a slice will be a *TextureRegion*:

```
explosion_tex_seq = image.TextureGrid(explosion_seq)
```

Because *TextureGrid* is also a *Texture*, you can use it either as individual images or as the whole grid at once.

3D textures

TextureGrid is extremely efficient for drawing many sprites from a single texture. One problem you may encounter, however, is bleeding between adjacent images.

When OpenGL renders a texture to the screen, by default it obtains each pixel colour by interpolating nearby texels. You can disable this behaviour by switching to the `GL_NEAREST` interpolation mode, however you then lose the benefits of smooth scaling, distortion, rotation and sub-pixel positioning.

You can alleviate the problem by always leaving a 1-pixel clear border around each image frame. This will not solve the problem if you are using mipmapping, however. At this stage you will need a 3D texture.

You can create a 3D texture from any sequence of images, or from an *ImageGrid*. The images must all be of the same dimension, however they need not be powers of two (pyglet takes care of this by returning *TextureRegion* as with a regular *Texture*).

In the following example, the explosion texture from above is uploaded into a 3D texture:

```
explosion_3d = pyglet.image.Texture3D.create_for_image_grid(explosion_seq)
```

You could also have stored each image as a separate file and used *Texture3D.create_for_images* to create the 3D texture.

Once created, a 3D texture behaves like any other *ImageSequence*; slices return *TextureRegion* for an image plane within the texture. Unlike a *TextureGrid*, though, you cannot blit a *Texture3D* in its entirety.

Texture bins and atlases

Image grids are useful when the artist has good tools to construct the larger images of the appropriate format, and the contained images all have the same size. However it is often simpler to keep individual images as separate files on disk, and only combine them into larger textures at runtime for efficiency.

A *TextureAtlas* is initially an empty texture, but images of any size can be added to it at any time. The atlas takes care of tracking the "free" areas within the texture, and of placing images at appropriate locations within the texture to avoid overlap.

It's possible for a *TextureAtlas* to run out of space for new images, so applications will need to either know the correct size of the texture to allocate initally, or maintain multiple atlases as each one fills up.

The *TextureBin* class provides a simple means to manage multiple atlases. The following example loads a list of images, then inserts those images into a texture bin. The resulting list is a list of *TextureRegion* images that map into the larger shared texture atlases:

```
images = [
    pyglet.image.load('img1.png'),
    pyglet.image.load('img2.png'),
    # ...
]

bin = pyglet.image.atlas.TextureBin()
images = [bin.add(image) for image in images]
```

The *pyglet.resource* module (see *Application resources*) uses texture bins internally to efficiently pack images automatically.

Animations

While image sequences and atlases provide storage for related images, they alone are not enough to describe a complete animation.

The *Animation* class manages a list of *AnimationFrame* objects, each of which references an image and a duration, in seconds. The storage of the images is up to the application developer: they can each be discrete, or packed into a texture atlas, or any other technique.

An animation can loaded directly from a GIF 89a image file with *load_animation* (supported on Linux, Mac OS X and Windows) or constructed manually from a list of images or an image sequence using the class methods (in which case the timing information will also need to be provided). The *add_to_texture_bin* method provides a convenient way to pack the image frames into a texture bin for efficient access.

Individual frames can be accessed by the application for use with any kind of rendering, or the entire animation can be used directly with a *Sprite* (see next section).

The following example loads a GIF animation and packs the images in that animation into a texture bin. A sprite is used to display the animation in the window:

```
animation = pyglet.image.load_animation('animation.gif')
bin = pyglet.image.TextureBin()
animation.add_to_texture_bin(bin)
```

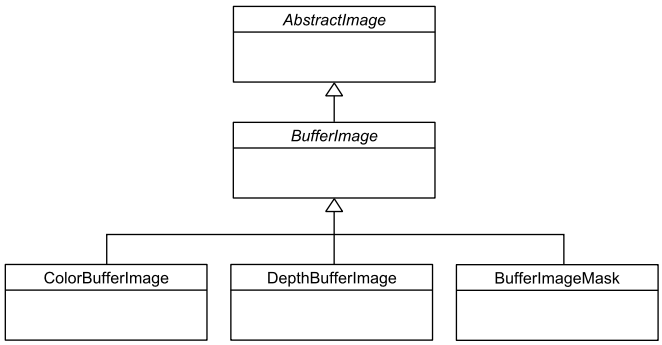
```
sprite = pyglet.sprite.Sprite(animation)

window = pyglet.window.Window()

@window.event
def on_draw():
    sprite.draw()

pyglet.app.run()
```

When animations are loaded with *pyglet.resource* (see *Application resources*) the frames are automatically packed into a texture bin.

This example program is located in *examples/programming_guide/animation.py*, along with a sample GIF animation file.

Buffer images

pyglet provides a basic representation of the framebuffer as components of the *AbstractImage* hierarchy. At this stage this representation is based off OpenGL 1.1, and there is no support for newer features such as framebuffer objects. Of course, this doesn't prevent you using framebuffer objects in your programs -- *pyglet.gl* provides this functionality -- just that they are not represented as *AbstractImage* types.



The *BufferImage* hierarchy.

A framebuffer consists of

- One or more colour buffers, represented by *ColorBufferImage*
- An optional depth buffer, represented by *DepthBufferImage*
- An optional stencil buffer, with each bit represented by *BufferImageMask*
- Any number of auxilliary buffers, also represented by *ColorBufferImage*

You cannot create the buffer images directly; instead you must obtain instances via the *BufferManager*. Use *get_buffer_manager* to get this singleton:

```
buffers = image.get_buffer_manager()
```

Only the back-left color buffer can be obtained (i.e., the front buffer is inaccessible, and stereo contexts are not supported by the buffer manager):

```
color_buffer = buffers.get_color_buffer()
```

This buffer can be treated like any other image. For example, you could copy it to a texture, obtain its pixel data, save it to a file, and so on. Using the *texture* attribute is particularly useful, as it allows you to perform multipass rendering effects without needing a render-to-texture extension.

The depth buffer can be obtained similarly:

```
depth_buffer = buffers.get_depth_buffer()
```

When a depth buffer is converted to a texture, the class used will be a *DepthTexture*, suitable for use with shadow map techniques.

The auxilliary buffers and stencil bits are obtained by requesting one, which will then be marked as "in-use". This permits multiple libraries and your application to work together without clashes in stencil bits or auxilliary buffer names. For example, to obtain a free stencil bit:

```
mask = buffers.get_buffer_mask()
```

The buffer manager maintains a weak reference to the buffer mask, so that when you release all references to it, it will be returned to the pool of available masks.

Similarly, a free auxilliary buffer is obtained:

```
aux_buffer = buffers.get_aux_buffer()
```

When using the stencil or auxilliary buffers, make sure you explicitly request these when creating the window. See *OpenGL configuration options* for details.

Displaying images

Images should be drawn into a window in the window's *on_draw* event handler. Usually a "sprite" should be created for each appearance of the image on-screen. Images can also be drawn directly without creating a sprite.

Sprites

A sprite is an instance of an image displayed in the window. Multiple sprites can share the same image; for example, hundreds of bullet sprites might share the same bullet image.

A sprite is constructed given an image or animation, and drawn with the *Sprite.draw* method:

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

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

Sprites have properties for setting the position, rotation, scale, opacity, color tint and visibility of the displayed image. Sprites automatically handle displaying the most up-to-date frame of an animation. The following example uses a scheduled function to gradually move the sprite across the screen:

```
def update(dt):
    # Move 10 pixels per second
    sprite.x += dt * 10

# Call update 60 times a second
```

```
pyglet.clock.schedule_interval(update, 1/60.)
```

If you need to draw many sprites, use a *Batch* to draw them all at once. This is far more efficient than calling *draw* on each of them in a loop:

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

sprites = [pyglet.sprite.Sprite(image, batch=batch),
           pyglet.sprite.Sprite(image, batch=batch),
           # ...  ]

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

When sprites are collected into a batch, no guarantee is made about the order in which they will be drawn. If you need to ensure some sprites are drawn before others (for example, landscape tiles might be drawn before character sprites, which might be drawn before some particle effect sprites), use two or more *OrderedGroup* objects to specify the draw order:

```
batch = pyglet.graphics.Batch()
background = pyglet.graphics.OrderedGroup(0)
foreground = pyglet.graphics.OrderedGroup(1)

sprites = [pyglet.sprite.Sprite(image, batch=batch, group=background),
           pyglet.sprite.Sprite(image, batch=batch, group=background),
           pyglet.sprite.Sprite(image, batch=batch, group=foreground),
           pyglet.sprite.Sprite(image, batch=batch, group=foreground),
           # ...]

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

See the *Graphics* section for more details on batch and group rendering.

For best performance, try to collect all batch images into as few textures as possible; for example, by loading images with *pyglet.resource.image* (see *Application resources*) or with *Texture bins and atlases*).

Simple image blitting

A simple but less efficient way to draw an image directly into a window is with the *blit* method:

```
@window.event
def on_draw():
    window.clear()
    image.blit(x, y)
```

The *x* and *y* coordinates locate where to draw the anchor point of the image. For example, to center the image at `(x, y)`:

```
kitten.anchor_x = kitten.width // 2
kitten.anchor_y = kitten.height // 2
```

```
kitten.blit(x, y)
```

You can also specify an optional *z* component to the *blit* method. This has no effect unless you have changed the default projection or enabled depth testing. In the following example, the second image is drawn *behind* the first, even though it is drawn after it:

```
from pyglet.gl import *
glEnable(GL_DEPTH_TEST)

kitten.blit(x, y, 0)
kitten.blit(x, y, -0.5)
```

The default pyglet projection has a depth range of (-1, 1) -- images drawn with a z value outside this range will not be visible, regardless of whether depth testing is enabled or not.

Images with an alpha channel can be blended with the existing framebuffer. To do this you need to supply OpenGL with a blend equation. The following code fragment implements the most common form of alpha blending, however other techniques are also possible:

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

You would only need to call the code above once during your program, before you draw any images (this is not necessary when using only sprites).

OpenGL imaging

This section assumes you are familiar with texture mapping in OpenGL (for example, chapter 9 of the OpenGL Programming Guide [http://opengl.org/documentation/red_book/]).

To create a texture from any *AbstractImage*, call *get_texture*:

```
kitten = image.load('kitten.jpg')
texture = kitten.get_texture()
```

Textures are automatically created and used by *ImageData* when blitted. It is useful to use textures directly when aiming for high performance or 3D applications.

The *Texture* class represents any texture object. The *target* attribute gives the texture target (for example, `GL_TEXTURE_2D`) and *id* the texture name. For example, to bind a texture:

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

Texture dimensions

Implementations of OpenGL prior to 2.0 require textures to have dimensions that are powers of two (i.e., 1, 2, 4, 8, 16, ...). Because of this restriction, pyglet will always create textures of these dimensions (there are several non-conformant post-2.0 implementations). This could have unexpected results for a user blitting a texture loaded from a file of non-standard dimensions. To remedy this, pyglet returns a *TextureRegion* of the larger texture corresponding to just the part of the texture covered by the original image.

A *TextureRegion* has an *owner* attribute that references the larger texture. The following session demonstrates this:

```
>>> rgba = image.load('tests/image/rgba.png')
```

```
>>> rgba
<ImageData 235x257>         # The image is 235x257
>>> rgba.get_texture()
<TextureRegion 235x257>     # The returned texture is a region
>>> rgba.get_texture().owner
<Texture 256x512>           # The owning texture has power-2 dimensions
>>>
```

A *TextureRegion* defines a *tex_coords* attribute that gives the texture coordinates to use for a quad mapping the whole image. *tex_coords* is a 4-tuple of 3-tuple of floats; i.e., each texture coordinate is given in 3 dimensions. The following code can be used to render a quad for a texture region:

```
texture = kitten.get_texture()
t = texture.tex_coords
w, h = texture.width, texture.height
array = (GLfloat * 32)(
     t[0][0], t[0][1], t[0][2], 1.,
     x,       y,       z,       1.,
     t[1][0], t[1][1], t[1][2], 1.,
     x + w,   y,       z,       1.,
     t[2][0], t[2][1], t[2][2], 1.,
     x + w,   y + h,   z,       1.,
     t[3][0], t[3][1], t[3][2], 1.,
     x,       y + h,   z,       1.)

glPushClientAttrib(GL_CLIENT_VERTEX_ARRAY_BIT)
glInterleavedArrays(GL_T4F_V4F, 0, array)
glDrawArrays(GL_QUADS, 0, 4)
glPopClientAttrib()
```

The *Texture.blit* method does this.

Use the `Texture.create` method to create either a texture region from a larger power-2 sized texture, or a texture with the exact dimensions using the `GL_texture_rectangle_ARB` extension.

Texture internal format

pyglet automatically selects an internal format for the texture based on the source image's *format* attribute. The following table describes how it is selected.

Format	**Internal format**
Any format with 3 components	`GL_RGB`
Any format with 2 components	`GL_LUMINANCE_ALPHA`
`"A"`	`GL_ALPHA`
`"L"`	`GL_LUMINANCE`
`"I"`	`GL_INTENSITY`
Any other format	`GL_RGBA`

Note that this table does not imply any mapping between format components and their OpenGL counterparts. For example, an image with format `"RG"` will use `GL_LUMINANCE_ALPHA` as its internal format; the luminance channel will be averaged from the red and green components, and the alpha channel will be empty (maximal).

Use the `Texture.create` class method to create a texture with a specific internal format.

Saving an image

Any image can be saved using the *save* method:

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

or, specifying a file-like object:

```
kitten_stream = open('kitten.png', 'wb')
kitten.save('kitten.png', file=kitten_stream)
```

The following example shows how to grab a screenshot of your application window:

```
pyglet.image.get_buffer_manager().get_color_buffer().save('screenshot.png')
```

Note that images can only be saved in the PNG format unless PIL is installed.

Sound and video

pyglet can play many audio and video formats. Audio is played back with either OpenAL, DirectSound or ALSA, 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 [http://code.google.com/p/avbin], an optional component available for Linux, Windows and Mac OS X. AVbin is installed alongside pyglet by default if the Windows or Mac OS X installation is used. If pyglet was installed from source, AVbin can 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

pyglet can use OpenAL, DirectSound or ALSA to play back audio. Only one of these drivers can be used in an application, and this must be selected before the *pyglet.media* module is loaded. The available drivers depend on your operating system:

Windows	**Mac OS X**	**Linux**
OpenAL [11]	OpenAL	OpenAL [11]
DirectSound		
		ALSA

[11] OpenAL 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 [http://www.openal.org/downloads.html]

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

```
pyglet.options['audio'] = ('openal', 'silent')
```

This tells pyglet to use the OpenAL driver if it is available, and to ignore all audio output if it is not. The `audio` option can be a list of any of these strings, giving the preference order for each driver:

String	**Audio driver**
`openal`	OpenAL
`directsound`	DirectSound
`alsa`	ALSA
`silent`	No audio output

You must set the `audio` option before importing *pyglet.media*. You can alternatively set it through an environment variable; see *Environment settings*.

The following sections describe the requirements and limitations of each audio driver.

DirectSound

DirectSound is available only on Windows, and is installed by default on Windows XP and later. pyglet uses only DirectX 7 features. On Windows Vista DirectSound does not support hardware audio mixing or surround sound.

OpenAL

OpenAL is included with Mac OS X. Windows users can download a generic driver from openal.org [http://www.openal.org/downloads.html], 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 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.

ALSA

ALSA is the standard Linux audio implementation, and is installed by default with many distributions. Due to limitations in ALSA all audio sources will play back at full volume and without any surround sound positioning.

Linux Issues

Linux users have the option of choosing between OpenAL and ALSA for audio output. Unfortunately both implementations have severe limitations or implementation bugs that are outside the scope of pyglet's control.

If your application can manage without stereo playback, or needs control over individual audio volumes, you should use the OpenAL driver (assuming your users have it installed).

If your application needs stereo playback, or does not require spatialised sound, consider using the ALSA driver in preference to the OpenAL driver. You can do this with:

```
pyglet.options['audio'] = ('alsa', 'openal', 'silent')
```

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:

```
source = pyglet.media.load('explosion.wav')
```

If the media file is bundled with the application, consider using the resource module (see *Application resources*).

The result of loading a media file is a *Source* object. This object provides useful information about the type of media encoded in the file, and serves as an opaque object used for playing back the file (described in the next section).

The *load* function will raise a *MediaException* if the format is unknown. *IOError* may also be raised if the file could not be read from disk. Future versions of pyglet will also support reading from arbitrary file-like objects, however a valid filename must currently be given.

The length of the media file is given by the *duration* property, which returns the media's length in seconds.

Audio metadata is provided in the source's *audio_format* attribute, which is *None* for silent videos. This metadata is not generally useful to applications. See the *AudioFormat* class documentation for details.

Video metadata is provided in the source's *video_format* attribute, which is *None* for audio files. It is recommended that this attribute is checked before attempting play back a video file -- if a movie file has a readable audio track but unknown video format it will appear as an audio file.

You can use the video metadata, described in a *VideoFormat* object, to set up display of the video before beginning playback. The attributes are as follows:

Attribute	Description
`width`, `height`	Width and height of the video image, in pixels.
`sample_aspect`	The aspect ratio of each video pixel.

You must take care to apply the sample aspect ratio to the video image size for display purposes. The following code determines the display size for a given video format:

```
def get_video_size(width, height, sample_aspect):
    if sample_aspect > 1.:
        return width * sample_aspect, height
    elif sample_aspect < 1.:
        return width, height / sample_aspect
    else:
        return width, height
```

Media files are not normally read entirely from disk; instead, they are streamed into the decoder, and then into the audio buffers and video memory only when needed. This reduces the startup time of loading a file and reduces the memory requirements of the application.

However, there are times when it is desirable to completely decode an audio file in memory first. For example, a sound that will be played many times (such as a bullet or explosion) should only be decoded once. You can instruct pyglet to completely decode an audio file into memory at load time:

```
explosion = pyglet.media.load('explosion.wav', streaming=False)
```

The resulting source is an instance of *StaticSource*, which provides the same interface as a streaming source. You can also construct a *StaticSource* directly from an already-loaded *Source*:

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

Simple audio playback

Many applications, especially games, need to play sounds in their entirety without needing to keep track of them. For example, a sound needs to be played when the player's space ship explodes, but this sound never needs to have its volume adjusted, or be rewound, or interrupted.

pyglet provides a simple interface for this kind of use-case. Call the *play* method of any *Source* to play it immediately and completely:

```
explosion = pyglet.media.load('explosion.wav', streaming=False)
explosion.play()
```

You can call *play* on any *Source*, not just *StaticSource*.

The return value of *Source.play* is a *ManagedPlayer*, which can either be discarded, or retained to maintain control over the sound's playback.

Controlling playback

You can implement many functions common to a media player using the *Player* class. Use of this class is also necessary for video playback. There are no parameters to its construction:

```
player = pyglet.media.Player()
```

A player will play any source that is "queued" on it. Any number of sources can be queued on a single player, but once queued, a source can never be dequeued (until it is removed automatically once complete). The main use of this queuing mechanism is to facilitate "gapless" transitions between playback of media files.

A *StreamingSource* can only ever be queued on one player, and only once on that player. *StaticSource* objects can be queued any number of times on any number of players. Recall that a *StaticSource* can be created by passing `streaming=False` to the *load* method.

In the following example, two sounds are queued onto a player:

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

Playback begins with the player's *play* method is called:

```
player.play()
```

Standard controls for controlling playback are provided by these methods:

Method	**Description**
play	Begin or resume playback of the current source.
pause	Pause playback of the current source.
next	Dequeue the current source and move to the next one immediately.
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* method does not guarantee gapless playback.

There are several properties that describe the player's current state:

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

When a player reaches the end of the current source, by default it will move immediately to the next queued source. If there are no more sources, playback stops until another is queued. There are several other possible behaviours, specified by setting the *eos_action* attribute on the player:

`eos_action`	**Description**
EOS_NEXT	The default action: playback continues at the next source.
EOS_PAUSE	Playback pauses at the end of the source, which remains the current source for this player.
EOS_LOOP	Playback continues immediately at the beginning of the current source.
EOS_STOP	Valid only for *ManagedPlayer*, for which it is default: the player is discarded when the current source finishes.

You can change a player's *eos_action* 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 *eos_action* is 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 syncronised with the audio track, for example:

```
@window.event
def on_draw():
    player.get_texture().blit(0, 0)
```

The texture is an instance of *pyglet.image.Texture*, with an internal format of either `GL_TEXTURE_2D` or `GL_TEXTURE_RECTANGLE_ARB`. While the texture will typically be created only once and subsequentally 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 *pyglet.media.listener* singleton, an instance of *Listener*. This provides similar properties such as *Listener.position*, *Listener.forward_orientation* and *Listener.up_orientation* that describe the position of the user in 3D space.

Note that only mono sounds can be positioned. Stereo sounds will play back as normal, and only their volume and pitch properties will affect the sound.

Application resources

Previous sections in this guide have described how to load images, media and text documents using pyglet. Applications also usually have the need to load other data files: for example, level descriptions in a game, internationalised strings, and so on.

Programmers are often tempted to load, for example, an image required by their application with:

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

This code assumes `logo.png` is in the current working directory. Unfortunately the working directory is not necessarily the same as the directory containing the application script files.

- Applications started from the command line can start from an arbitrary working directory.
- Applications bundled into an egg, Mac OS X package or Windows executable may have their resources inside a ZIP file.
- The application might need to change the working directory in order to work with the user's files.

A common workaround for this is to construct a path relative to the script file instead of the working directory:

```
import os

script_dir = os.path.dirname(__file__)
path = os.path.join(script_dir, 'logo.png')
image = pyglet.image.load(path)
```

This, besides being tedious to write, still does not work for resources within ZIP files, and can be troublesome in projects that span multiple packages.

The *pyglet.resource* module solves this problem elegantly:

```
image = pyglet.resource.image('logo.png')
```

The following sections describe exactly how the resources are located, and how the behaviour can be customised.

Loading resources

Use the *pyglet.resource* module when files shipped with the application need to be loaded. For example, instead of writing:

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

use:

```
data_file = pyglet.resource.file('file.txt')
```

There are also convenience functions for loading media files for pyglet. The following table shows the equivalent resource functions for the standard file functions.

File function	Resource function	Type
`open`	*pyglet.resource.file*	File-like object

File function	Resource function	Type
pyglet.image.load	*pyglet.resource.image*	*Texture* or *TextureRegion*
pyglet.image.load	*pyglet.resource.texture*	*Texture*
pyglet.image.load_animation	*pyglet.resource.animation*	*Animation*
pyglet.media.load	*pyglet.resource.media*	*Source*
pyglet.text.load mimetype = `text/plain`	*pyglet.resource.text*	*UnformattedDocument*
pyglet.text.load mimetype = `text/html`	*pyglet.resource.html*	*FormattedDocument*
pyglet.text.load mimetype = `text/vnd.pyglet-attributed`	*pyglet.resource.attributed*	*FormattedDocument*
pyglet.font.add_file	*pyglet.resource.add_font*	None

pyglet.resource.texture is for loading stand-alone textures, and would be required when using the texture for a 3D model.

pyglet.resource.image is optimised for loading sprite-like images that can have their texture coordinates adjusted. The resource module attempts to pack small images into larger textures 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:

```
thumbnails_file = pyglet.resource.file('thumbnails.txt', 'rt')
thumbnails_location = pyglet.resource.location('thumbnails.txt')

for line in thumbnails_file:
    filename = line.strip()
    image_file = thumbnails_location.open(filename)
    image = pyglet.image.load(filename, file=image_file)
    # Do something with `image`...
```

This code correctly ignores other images with the same filename that might appear elsewhere on the resource path.

Specifying the resource path

By default, only the script home directory is searched (the directory containing the `__main__` module). You can set *pyglet.resource.path* to a list of locations to search in order. This list is indexed, so after modifying it you will need to call *pyglet.resource.reindex*.

Each item in the path list is either a path relative to the script home, or the name of a Python module preceded with an ampersand (@). For example, if you would like to package all your resources in a `res` directory:

```
pyglet.resource.path = ['res']
pyglet.resource.reindex()
```

Items on the path are not searched recursively, so if your resource directory itself has subdirectories, these need to be specified explicitly:

```
pyglet.resource.path = ['res', 'res/images', 'res/sounds', 'res/fonts']
pyglet.resource.reindex()
```

Specifying module names makes it easy to group code with its resources. The following example uses the directory containing the hypothetical `gui.skins.default` for resources:

```
pyglet.resource.path = ['@gui.skins.default', '.']
pyglet.resource.reindex()
```

Multiple loaders

A *Loader* encapsulates a complete resource path and cache. This lets your application cleanly separate resource loading of different modules. Loaders are constructed for a given search path, and exposes the same methods as the global *pyglet.resource* module functions.

For example, if a module needs to load its own graphics but does not want to interfere with the rest of the application's resource loading, it would create its own *Loader* with a local search path:

```
loader = pyglet.resource.Loader(['@' + __name__])
image = loader.image('logo.png')
```

This is particularly suitable for "plugin" modules.

You can also use a *Loader* instance to load a set of resources relative to some user-specified document directory. The following example creates a loader for a directory specified on the command line:

```
import sys
home = sys.argv[1]
loader = pyglet.resource.Loader(script_home=[home])
```

This is the only way that absolute directories and resources not bundled with an application should be used with *pyglet.resource*.

Saving user preferences

Because Python applications can be distributed in several ways, including within ZIP files, it is usually not feasible to save user preferences, high score lists, and so on within the application directory (or worse, the working directory).

The *pyglet.resource.get_settings_path* function returns a directory suitable for writing arbitrary user-centric data. The directory used follows the operating system's convention:

- `~/.ApplicationName/` on Linux
- `$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:

```
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')
```

Debugging tools

pyglet includes a number of debug paths that can be enabled during or before application startup. These were primarily developed to aid in debugging pyglet itself, however some of them may also prove useful for understanding and debugging pyglet applications.

Each debug option is a key in the *pyglet.options* dictionary. Options can be set directly on the dictionary before any other modules are imported:

```
import pyglet
pyglet.options['debug_gl'] = False
```

They can also be set with environment variables before pyglet is imported. The corresponding environment variable for each option is the string `PYGLET_` prefixed to the uppercase option key. For example, the environment variable for `debug_gl` is `PYGLET_DEBUG_GL`. Boolean options are set or unset with `1` and `0` values.

A summary of the debug environment variables appears in the table below.

Option	Environment variable	Type
`debug_font`	`PYGLET_DEBUG_FONT`	bool
`debug_gl`	`PYGLET_DEBUG_GL`	bool
`debug_gl_trace`	`PYGLET_DEBUG_GL_TRACE`	bool
`debug_gl_trace_args`	`PYGLET_DEBUG_GL_TRACE_ARGS`	bool
`debug_graphics_batch`	`PYGLET_DEBUG_GRAPHICS_BATCH`	bool
`debug_lib`	`PYGLET_DEBUG_LIB`	bool
`debug_media`	`PYGLET_DEBUG_MEDIA`	bool
`debug_trace`	`PYGLET_DEBUG_TRACE`	bool
`debug_trace_args`	`PYGLET_DEBUG_TRACE_ARGS`	bool
`debug_trace_depth`	`PYGLET_DEBUG_TRACE_DEPTH`	int
`debug_win32`	`PYGLET_DEBUG_WIN32`	bool
`debug_x11`	`PYGLET_DEBUG_X11`	bool
`graphics_vbo`	`PYGLET_GRAPHICS_VBO`	bool

The `debug_media` and `debug_font` options are used to debug the `pyglet.media` and `pyglet.font` modules, respectively. Their behaviour is platform-dependent and useful only for pyglet developers.

The remaining debug options are detailed below.

Debugging OpenGL

The `graphics_vbo` option enables the use of vertex buffer objects in *pyglet.graphics* (instead, only vertex arrays). This is useful when debugging the `graphics` module as well as isolating code for determining if a video driver is faulty.

The `debug_graphics_batch` option causes all *Batch* objects to dump their rendering tree to standard output before drawing, after any change (so two drawings of the same tree will only dump once). This is useful to debug applications making use of *Group* and *Batch* rendering.

Error checking

The `debug_gl` option intercepts most OpenGL calls and calls `glGetError` afterwards (it only does this where such a call would be legal). If an error is reported, an exception is raised immediately.

This option is enabled by default unless the `-O` flag (optimisation) is given to Python, or the script is running from within a py2exe or py2app package.

Tracing

The `debug_gl_trace` option causes all OpenGL functions called to be dumped to standard out. When combined with `debug_gl_trace_args`, the arguments given to each function are also printed (they are abbreviated if necessary to avoid dumping large amounts of buffer data).

Tracing execution

The `debug_trace` option enables Python-wide function tracing. This causes every function call to be printed to standard out. Due to the large number of function calls required just to initialise pyglet, it is recommended to redirect standard output to a file when using this option.

The `debug_trace_args` option additionally prints the arguments to each function call.

When `debug_trace_depth` is greater than 1 the caller(s) of each function (and their arguments, if `debug_trace_args` is set) are also printed. Each caller is indented beneath the callee. The default depth is 1, specifying that no callers are printed.

Platform-specific debugging

The `debug_lib` option causes the path of each loaded library to be printed to standard out. This is performed by the undocumented `pyglet.lib` module, which on Linux and Mac OS X must sometimes follow complex procedures to find the correct library. On Windows not all libraries are loaded via this module, so they will not be printed (however, loading Windows DLLs is sufficiently simple that there is little need for this information).

Linux

X11 errors are caught by pyglet and suppressed, as there are plenty of X servers in the wild that generate errors that can be safely ignored. The `debug_x11` option causes these errors to be dumped to standard out, along with a traceback of the Python stack (this may or may not correspond to the error, depending on whether or not it was reported asynchronously).

Windows

The `debug_win32` option causes all library calls into `user32.dll`, `kernel32.dll` and `gdi32.dll` to be intercepted. Before each library call `SetLastError(0)` is called, and afterwards `GetLastError()` is called. Any errors discovered are written to a file named `debug_win32.log`. Note that an error is only valid if the function called returned an error code, but the interception function does not check this.

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

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:

```
from pyglet import font
from pyglet import image
from pyglet import window
```

pyglet now lazily loads submodules on demand, so an application can get away with importing just *pyglet*. This is especially handy for modules that are typically only used once in an application, and frees up the names `font`, `image`, `window` and so on for the application developer. For example:

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

Application event loop

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

```
while not window.has_exit:
    dt = clock.tick()
    update(dt)

    window.dispatch_events()
    window.clear()
    draw()
    window.flip()
```

Besides being somewhat repetitious to type, this type of event loop is difficult to extend with more windows, and exausts all available system resources, even if the application is not doing anything.

The new *pyglet.app* module provides an application event loop that is less demanding of the CPU yet more responsive to user events. A complete application that opens an empty window can be written with:

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

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

pyglet.app.run()
```

Note the new *on_draw* event, which makes it easy to specify different drawing functions for each window. The *pyglet.app* event loop takes care of dispatching events, ticking the clock, calling the draw function and flipping the window buffer.

Update functions can be scheduled on the clock. To have an update function be called as often as possible, use *clock.schedule* (this effectively degenerates into the older *dispatch_events* practice of thrashing the CPU):

```
def update(dt):
    pass
clock.schedule(update)
```

Usually applications can update at a less frequent interval. For example, a game that is designed to run at 60Hz can use *clock.schedule_interval*:

```
def update(dt):
    pass
clock.schedule_interval(update, 1/60.0)
```

This also removes the need for *clock.set_fps_limit*.

Besides the advantages already listed, windows managed by the event loop will not block while being resized or moved; and the menu bar on OS X can be interacted with without blocking the application.

It is highly recommended that all applications use the event loop. The loop can be extended if you need to add additional hooks or integrate with another package. Applications continuing to use *Window.dispatch_events* gain no advantage, but suffer from poorer response, increased CPU usage and artifacts during window resizing and moving.

See *The application event loop* for more details.

Loading resources

Locating resources such as images, sound and video files, data files and fonts is difficult to do correctly across all platforms, considering the effects of a changing working directory and various distribution packages such as setuptools, py2exe and py2app.

The new *pyglet.resource* module implements the correct logic for all these cases, making it simple to load resources that belong to a specific module or the application as a whole. A resource path can be set that is indexed once, and can include filesystem directories, Python module paths and ZIP files.

For example, suppose your application ships with a `logo.png` that needs to be loaded on startup. In pyglet 1.0 you might have written:

```
import os.path
from pyglet import image

script_dir = os.path.dirname(__file__)
logo_filename = os.path.join(script_dir, 'logo.png')
logo = image.load(logo_filename)
```

In pyglet 1.1, you can write:

```
logo = pyglet.resource.image('logo.png')
```

And will actually work in more scenarios (such as within a setuptools egg file, py2exe and py2app).

The resource module efficiently packs multiple small images into larger textures, so there is less need for artists to create sprite sheets themselves for efficient rendering. Images and textures are also cached automatically.

See *Application resources* for more details.

New graphics features

The *pyglet.graphics* module is a low-level abstraction of OpenGL vertex arrays and buffer objects. It is intended for use by developers who are already very familiar with OpenGL and are after the best performance possible. pyglet uses this module internally to implement its new sprite module and the new text rendering module. The *Graphics* chapter describes this module in detail.

The *pyglet.sprite* module provide a fast, easy way to display 2D graphics on screen. Sprites can be moved, rotated, scaled and made translucent. Using the *batch* features of the new graphics API, multiple sprites can be drawn in one go very quickly. See *Sprites* for details.

The *pyglet.image.load_animation* function can load animated GIF images. These are returned as an *Animation*, which exposes the individual image frames and timings. Animations can also be played directly on a sprite in place of an image. The *Animations* chapter describes how to use them.

The *pyglet.image.atlas* module packs multiple images into larger textures for efficient rendering. The *pyglet.resource* module uses this module for small images automatically, but you can use it directly even if you're not making use of *pyglet.resource*. See *Texture bins and atlases* for details.

Images now have `anchor_x` and `anchor_y` attributes, which specify a point from which the image should be drawn. The sprite module also uses the anchor point as the center of rotation.

Textures have a *get_transform* method for retrieving a *TextureRegion* that refers to the same texture data in video memory, but with optional horizontal or vertical flipping, or 90-degree rotation.

New text features

The *pyglet.text* module can render formatted text efficiently. A new class *Label* supercedes the old *pyglet.font.Text* class (which is now actually implemented in terms of *Label*). The "Hello, World" application can now written:

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

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

pyglet.app.run()
```

You can also display multiple fonts and styles within one label, with *HTMLLabel*:

```
label = pyglet.text.HTMLLabel('<b>Hello</b>, <font color=red>world!')
```

More advanced uses of the new text module permit applications to efficiently display large, scrolling, formatted documents (for example, HTML files with embedded images), and to allow the user to interactively edit text as in a WYSIWYG text editor.

Other new features

EventDispatcher now has a *remove_handlers* method which provides finer control over the event stack than *pop_handlers*.

The `@event` decorator has been fixed so that it no longer overrides existing event handlers on the object, which fixes the common problem of handling the *on_resize* event. For example, the following now works without any surprises (in pyglet 1.0 this would override the default handler, which sets up a default, necessary viewport and projection):

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

```
pass
```

A variant of *clock.schedule_interval*, *clock.schedule_interval_soft* has been added. This is for functions that need to be called periodically at a given interval, but do not need to schedule the period immediately. Soft interval scheduling is used by the *pyglet.media* module to distribute the work of decoding video and audio data over time, rather than stalling the CPU periodically. Games could use soft interval scheduling to spread the regular computational requirements of multiple agents out over time.

In pyglet 1.0, *font.load* attempted to match the font resolution (DPI) with the operating system's typical behaviour. For example, on Linux and Mac OS X the default DPI was typically set at 72, and on Windows at 96. While this would be useful for writing a word processor, it adds a burden on the application developer to ensure their fonts work at arbitrary resolutions. In pyglet 1.1 the default DPI is set at 96 across all platforms. It can still be overridden explicitly by the application if desired.

Video sources in *pyglet.media* can now be stepped through frame-by-frame: individual image frames can be extracted without needing to play back the video in realtime.

For a complete list of new features and bug fixes, see the `CHANGELOG` distributed with the source distribution.