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*.

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-1.0alpha1\;%PYTHONPATH%
```

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

```
set PYTHONPATH /path/to/pyglet-1.0alpha1/:$PYTHONPATH
export PYTHONPATH
```

or, using tcsh or a variant:

```
setenv PYTHONPATH /path/to/pyglet-1.0alpha1/:$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-1.0alpha1/`.

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 modules from pyglet that we need:

```
from pyglet import font
from pyglet import window
```

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:

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

To display the text, we'll create a *font.Text*. The text requires a font, which we'll load from the system:

```
ft = font.load('Arial', 36)
text = font.Text(ft, 'Hello, World!')
```

Every pyglet application requires a "run loop". You'll always need to write this loop yourself. Within the loop, we need to:

- Update events for the window. If you forget to do this, the window will appear to "hang" and be unresponsive.
- Clear the window
- Draw the contents of the window.

The run loop is shown below:

```
while not win.has_exit:
    win.dispatch_events()
    win.clear()
    text.draw()
    win.flip()
```

By default, windows in pyglet are double-buffered, which is necessary for flicker-free animation. The *flip* method makes all previous drawing operations visible.

Image viewer

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

```
from pyglet import image
from pyglet import window
```

```
win = window.Window()

img = image.load('kitten.jpg')

while not win.has_exit:
    win.dispatch_events()
    win.clear()
    img.blit(0, 0)
    win.flip()
```

We used the *image.load* function to load the image, and *AbstractImage.blit* method to draw it to the window. 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

The `dispatch_events` method seen in the previous examples not only takes care of keeping the application responsive, it also calls any event handlers you install on the window.

There are several ways to attach an event; the most straight-forward is to simply replace the method on the *Window* with your own function:

```
from pyglet import window

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

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

Remember to also write a run loop that calls *dispatch_events* every iteration.

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

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
```

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

win.on_mouse_press = on_mouse_press
```

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:

```
from pyglet.window import event

win.push_handlers(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]:

```
from pyglet import media

music = media.load('music.mp3')
music.play()
```

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

Regardless of whether you are using streaming or non-streaming sounds, you must call *media.dispatch_events* in your run loop, in the same way that you do for *Window.dispatch_events*:

```
while not win.has_exit:
    win.dispatch_events()
    media.dispatch_events()
```

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

Where to next?

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

[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.

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.

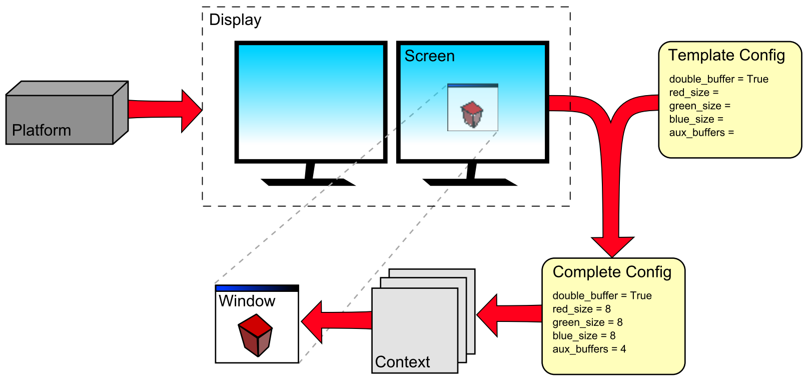
To write 3D applications or achieve optimal performance in your 2D applications you'll need to work with OpenGL directly. The canonical references for OpenGL are The OpenGL Programming Guide [http://opengl.org/documentation/books/#the_opengl_programming_guide_the_official_guide_to_learning_opengl_version]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

```
>>> from pyglet import window
>>> win = window.Window()
>>> context = win.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 = 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

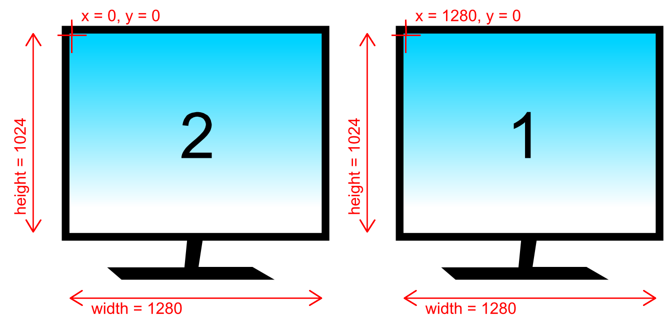
Once you have obtained a display, you can enumerate the screens that are connected. A screen is the physical display medium connected to the display device; for example a computer monitor, TV or projector. Most computers will have a single screen, however dual-head workstations and laptops connected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

```
from pyglet import gl
from pyglet import window

config = gl.Config(alpha_size=8)
win = 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 = window.get_platform()
display = platform.get_default_display()
screen = display.get_default_screen()

template = gl.Config(alpha_size=8)
config = screen.get_best_config(template)
context = config.create_context(None)
win = win.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 window.NoSuchConfigException:
    template = gl.Config()
    config = screen.get_best_config(template)
win = win.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:

```
from pyglet import gl
from pyglet import window

platform = 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.

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:

```
from pyglet import gl
```

```
context = 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 *
from pyglet import window

# The OpenGL context is created only when you create a window.  Any calls
# to OpenGL before a window is created will fail.
win = window.Window(visible=False)

# ... perform any OpenGL state initialisation here.

win.set_visible()
while not win.has_exit:
    win.dispatch_events()

    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glBegin(GL_TRIANGLES)
    glVertex2f(0, 0)
    glVertex2f(win.width, 0)
    glVertex2f(win.width, win.height)
    glEnd()

    win.flip()
```

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 *
from pyglet import window

win = window.Window(visible=False)

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)

win.set_visible()
while not win.has_exit:
    win.dispatch_events()

    glClear(GL_COLOR_BUFFER_BIT)
    glLoadIdentity()
    glDrawArrays(GL_TRIANGLES, 0, len(vertices) // 2)

    win.flip()
```

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*:

```
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:

```
def on_resize(width, height):
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluPerspective(65, width / float(height), .1, 1000)
    glMatrixMode(GL_MODELVIEW)
```

```
win.on_resize = on_resize
```

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
from pyglet import options
options['debug_gl'] = False

from pyglet.gl import *
```

Setting the option after importing `pyglet.gl` will have no effect. Once disabled, there is no 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:

```
from pyglet.gl import gl_info

if 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 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 [7]. Each context has its own state and framebuffers.

There is always an active context (unless there are no windows). By default, the active context belongs to whichever window was displayed last. This works well if there is only ever one window visible at a time. If there are more, you need to explicitly set the active context when you start drawing. You can do this by calling the window's `switch_to` method:

```
window_1 = window.Window()
window_2 = window.Window()

while running:
    window_1.dispatch_events()
    window_2.dispatch_events()

    # Draw the contents of window_1:
    window_1.switch_to()
    glClear(GL_COLOR_BUFFER_BIT)
    # ...
    window_1.flip()

    window_2.switch_to()
    glClear(GL_COLOR_BUFFER_BIT)
    # ...
    window_2.flip()
```

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.

[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.

Similarly, if using WGL extensions, import `pyglet.gl.wglext_arb` or `pyglet.gl.wglext_nv`.

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.

pyglet does not manage its own run loop or operate any separate threads for the window, so you must ensure the operating system's event queue is cleared regularly. You do this by calling *Window.dispatch_events*:

```
from pyglet import window

win = window.Window()
while not win.has_exit:
    win.dispatch_events()
```

If you fail to call *dispatch_events*, the application will be unresponsive, and after some time the operating system may assume the program has crashed. If you create more than one window, you must call *dispatch_events* on each window.

The *has_exit* flag on each window is a convenience property that determines if the window has been closed by the user. This is managed by *WindowExitHandler*, which watches for the *on_close* event and *on_key_press* event, and sets *has_exit* to True if the close box is clicked or the escape key is pressed.

If you write your own handler for either of these events without passing control onto the next handler in line, the *has_exit* flag will not be set. This can be useful as it allows you to override this default functionality. For example, you could ask the user if they wish to save their data before closing the window.

Creating a window

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

```
from pyglet import window

win = 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. This can be useful for attaching event handlers before the window is displayed. The following example shows how to create and display a window in two steps:

```
win = window.Window(visible=False)
# ... perform some additional initialisation on the window
win.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)
  win = 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)
  win = 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)
  win = 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]
  win = 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)
  win = 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()
win = 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:

```
win.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.

```
from pyglet import window

win = 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:

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

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

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:

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

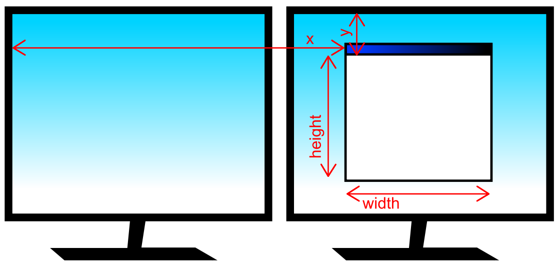
The window can also be resized programatically (even if the window is not user-resizable) with the *set_size* method:

```
win.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 = win.get_location()
win.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:

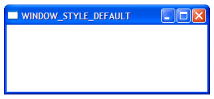
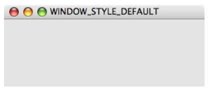
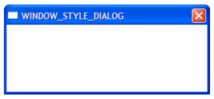
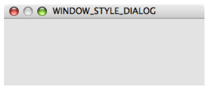
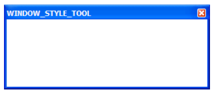
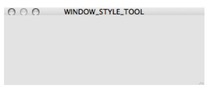


The position ond size of the window relative to the desktop.

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:

```
from pyglet import window

win = 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:

```
win = window.Window(caption='Initial caption')
win.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:

```
from pyglet import image
from pyglet import window

win = window.Window()
icon1 = image.load('16x16.png')
icon2 = image.load('32x32.png')
win.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. As well as being set in the constructor, you can change or read this after the window is created:

```
>>> from pyglet import window
>>> win = window.Window(visible=False)
>>> win.visible
False
>>> win.set_visible(True)
```

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*:

```
from pyglet import font
from pyglet import window

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

        ft = font.load('Arial', 36)
        self.text = font.Text(ft, 'Hello, World!')

    def draw(self):
        self.text.draw()

    def run(self):
        while not self.has_exit:
            self.dispatch_events()

            self.clear()
            self.draw()
            self.flip()

if __name__ == '__main__':
    HelloWorldWindow().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:

```
from pyglet import window

win = window.Window()
win.switch_to()

# .. Perform OpenGL drawing operations

win.flip()
```

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:

```
win = 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 pyglet event framework

Both the *pyglet.window* and *pyglet.media* 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:

```
from pyglet import window

win = window.Window()

def on_resize(width, height):
    pass
win.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:

```
win = window.Window()

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

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

```
@win.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(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.

Dispatching events

When do event handlers get called? pyglet is single-threaded, and will only call event handlers at predictable points in your code. Events will queue up until a *dispatch_events* function or method is called, at which point the appropriate event handlers are called, in the order that the events arrived in.

Window.dispatch_events is the method which performs this task for a single window. Though the user may resize a window at any time, the *on_resize* handler shown above is only notified when you call *dispatch_events*. This leads inevitably to requiring some sort of main event or "run" loop:

```
while not win.has_exit:
    win.dispatch_events()
```

Note that unlike in other window frameworks, pyglet does not block on *dispatch_events* -- if there are no events pending, control is returned to your program immediately.

For *pyglet.media*, the single *media.dispatch_events* function handles the dispatch of events for all media instances. An application with two windows and any number of sounds and videos would have a run loop similar to:

```
from pyglet import media

while not (win1.has_exit or win2.has_exit):
    win1.dispatch_events()
    win2.dispatch_events()
    media.dispatch_events()
```

It is necessary to call *dispatch_events* for every window that you create, and for *pyglet.media* if you have any sounds or videos playing, even if you have no event handlers. This is so that behind-the-scenes operating system events can be processed (remember, pyglet is single-threaded).

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()

win.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

win.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

    win.push_handlers(on_key_press, on_mouse_press)

def end_game():
    win.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()
    win.push_handlers(game_handlers)

def stop_game()
    win.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.

It is not necessary to create a *dispatch_events* method; you may choose to dispatch events as soon as they are generated, or dispatch them from a separate thread. If you were to follow the pyglet model you would queue events until a *dispatch_events* method is called.

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

```
from pyglet import event

class ClankingWidget(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.'

register_event_type(ClankingWidget, 'on_clank')
register_event_type(ClankingWidget, '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, and only an additional attribute per event (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, *MediaInstance* is an *EventDispatcher* even though potentially hundreds of these objects may be created and destroyed each second, and most will not need an event handler.

Implementing the Observer pattern

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

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

```
# The subject
class ClockTimer(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). To document an event, create a method with the event's signature and add a blank `event` field to the docstring:

```
class MyDispatcher(object):
    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_BORDLESS* 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
```

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 import window
from pyglet.window import key

win = window.Window()
keys = key.KeyStateHandler()
win.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.

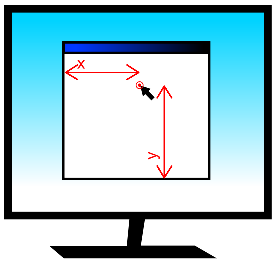
Working with the mouse

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*:

```
from pyglet import window
win = window.Window()
win.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 = win.get_system_mouse_cursor(win.CURSOR_HELP)
win.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	I	I
CURSOR_WAIT	⌛	⌚

Constant	**Windows XP**	**Mac OS X**
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:

```
from pyglet import image

img = image.load('cursor.png')
cursor = window.ImageMouseCursor(img, 16, 8)
win.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*:

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

Keeping track of time

For simple applications using animation, it is usually sufficient to use Python's built-in *time* module. pyglet's *clock* module provides additional functionality for writing low-latency run loops which do not waste CPU cycles, for scheduling functions for future execution in the same thread, and for displaying an attractive frame-rate counter for users.

Determining how much time has passed

Most pyglet applications will use a run loop similar to the following:

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

while not win.has_exit:
    win.dispatch_events()

    # ... update objects
    # ... render objects
    win.flip()
```

During the "update" phase, objects are moved according to their momentum and according to the user's input. The movement of an object is usually governed by simple discrete integration following Newtonian dynamics, i.e.:

```
velocity += time_passed * acceleration
position += time_passed * velocity
```

The *time_passed* variable is the amount of time since the last time the object was updated. pyglet's *clock.tick* function provides an easy way to calculate this value, by inserting it into the run loop as follows:

```
from pyglet import clock

while not win.has_exit:
    win.dispatch_events()

    time_passed = clock.tick()
    # ... update objects according to time_passed
    # ... render objects
    win.flip()
```

You must take care to call *tick* exactly once per frame. The value returned is a float giving the number of seconds that have passed since the last time *tick* was called.

Animation techniques

With the knowledge that your unit of time increments is known, you can specify complete units for all your other animation variables. Some examples are shown in the table below.

Animation parameter	**Distance**	**Velocity**
Rotation	Degrees	Degrees per second
Position	Pixels	Pixels per second

Animation parameter	**Distance**	**Velocity**
Keyframes	Frame number	Frames per second

For example, the code to move a sprite across the screen so that it covers 200 pixels every second is:

```
velocity = 200
position = 0

while not win.has_exit:
    win.dispatch_events()

    dt = clock.tick()
    position += dt * velocity

    # ... draw sprite at `position`
    win.flip()
```

Note the use of *dt* instead of *time_passed* -- this is a common naming convention within pyglet examples. It is an abbreviation of "delta (change in) time", as used in physics courses.

In the following example a list of images needs to be displayed in turn (like a flip book), with 10 images being displayed every second:

```
images = [image1, image2, image3, ...]
image_index = 0
image_period = 1.0 / 10             # Reciprocal of the frame rate
image_timeout = image_period

while not win.has_exit:
    win.dispatch_events()

    dt = clock.tick()
    image_timeout -= dt
    if image_timeout <= 0:
        image_index += 1
        image_timeout = image_period

    # ... draw `images[image_index]`
    win.flip()
```

Writing your animation or game code in this way ensures that it will play exactly the way you intend it to on all computers. On slower computers the animation may become "jerky", but it will move at the same rate. On faster computers the animation will be exceptionally smooth, but still move at the same rate.

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. If you are using the *clock.tick* function described in the previous section, you can determine your application's FPS with a single function call:

```
from pyglet import clock

clock.get_fps()
```

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

Controlling frame rate

If you have no need for high-speed animation, you may wish to limit your application's demand on the CPU. With *clock.set_fps_limit* you specify a target FPS, which pyglet will then try to match. For example, if you decide that it's sufficient to update the display only ten times a second:

```
clock.set_fps_limit(20)

while not win.has_exit:
    win.dispatch_events()

    dt = clock.tick()
    # ... update and render
    win.flip()
```

You must remember to call *tick* even if you are not using the return value. pyglet will yield control of the process back to the operating system if there is time to spare. It uses higher-resolution techniques than the built-in *time.sleep* function, which enables it to more accurately achieve the target frame rate. If the computer is not fast enough to render your target frame rate, *set_fps_limit* will have no effect.

Note that by default *Window.flip* is synchronised to the display refresh rate, which effectively caps your frame rate at 60-80 FPS, depending on the operating system display settings. See *Windows and OpenGL contexts* for more details.

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:

```
from pyglet import clock

fps_display = clock.ClockDisplay()

while not win.has_exit:
    # ...
    fps_display.draw()
    win.flip()
```

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.

Scheduling functions for future execution

Animation code is often simplified if you can write commands like, "in 5 seconds, start moving the umbrella," or, "change the colour of the player's face every 0.75 seconds."

The approach shown in *Animation techniques* can be used to implement this, but too many separately scheduled actions can quickly make the code complex and difficult to read, requiring many variables. It can also be inefficient: do you really want to be checking if a timeout scheduled for 5 minutes has elapsed every frame?

pyglet has a scheduler you can use to call functions at a later time. For example:

```
from pyglet import clock

# The function to call after 5 seconds
def move_umbrella(dt):
    pass

clock.schedule_once(move_umbrella, 5.0)
```

The *schedule_once* function ensures that the provided function is called after a certain amount of time has passed. Because it is impossible to call the function after *exactly* 5 seconds, the *dt* parameter gives the number of seconds that actually elapsed (for example, it may be 5.001).

You can use *schedule_interval* to have a function called periodically. For example:

```
# The function that is called every 0.75 seconds
def change_face_color(dt):
    pass

clock.schedule_interval(change_face_color, 0.75)
```

In this case the *dt* parameter gives the number of seconds since the last time the function was called, or since it was scheduled, if it is the first time (for example, 0.751).

To stop the scheduled action taking place from either *schedule_once* or *schedule_interval*, use *unschedule*:

```
clock.unschedule(move_umbrella)
clock.unschedule(change_face_color)
```

You should plan to do this if the animation needs to be aborted; for example, if the user starts the animation but then returns to the main menu, you need to unschedule all the functions already scheduled.

Scheduled functions are called during the *tick* function call; no separate threads are used. The clock uses an efficient priority queue giving linear cost with respect to the average number of scheduled functions that need to be called each frame, not the total number of scheduled functions. This means there is no performance penalty for scheduling lots of functions well in advance of when they need to be called.

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 text efficiently. Any installed font on the operating system can be used, or you can supply your own font with your application.

Some basic functionality is provided for rendering text in a single font, and additional complex rendering can be achieved by layering more sophisticated behaviour above the pyglet base classes.

Before you can load any fonts you must have an active OpenGL context. That is, you should create a window before loading fonts. If you need to load the font before you can determine the size of the window, set the visibility of the window to False initially:

```
from pyglet import font
from pyglet import window

win = win.Window(visible=False)

# ... load fonts and calculate window layout

win.set_size(width, height)
win.set_visible(True)
```

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:

```
from pyglet import font

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

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

```
times_bold = font.load('Times New Roman', 16, bold=True)
times_italic = font.load('Times New Roman', 16, italic=True)
times_bold_italic = 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 = 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 = 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. Each operating system defines its own assumed font resolution, given in dots (pixels) per inch (DPI):

Operating system	**Default DPI**
Windows XP	96, or 120 with "Large Fonts" enabled
Mac OS X	72
Linux	Depends on X11 configuration, but typically 96.

You should keep in mind when writing your application that fonts are displayed at different pixel sizes on different operating systems. For a typical application, this means you should not assume anything about the size of a font, except by using the font metrics functions desribed later.

This can be quite inconvenient when writing a graphics-intensive application such as a game, for example. If you need to mix text and graphics which cannot be easily resized, you can force pyglet to use a different font resolution. This is given as the *dpi* parameter in the *font.load* method and can be set differently for each font loaded:

```
from pyglet import font

times = font.load('Times New Roman', 12, dpi=96)
```

So, if you are primarily developing your application on Windows but wanted to ensure the fonts are displayed as similarly as possible on Mac OS X, you would specify a DPI of 96 when loading the font. Conversely, if you primarily develop on Mac OS X you would specify a DPI of 72.

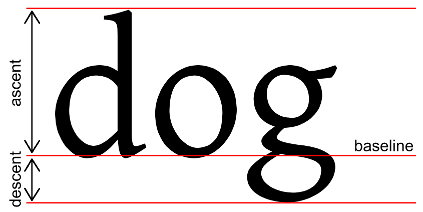
By specifying a DPI you are overriding both the operating system defaults and potentially any user-defined settings. This is not recommended for general-purpose applications, which should remain accessible to all users by not specifying a font resolution.

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:

```
from pyglet import font

font.add_file('action_man.ttf')
action_man = 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 *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.

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

Font Format	**Windows XP**	**Mac OS X**	**Linux (FreeType)**
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.

Basic text rendering

Most of the text requirements of applications written with pyglet can be taken care of with the *Text* class. This permits simple but efficient rendering of one or more lines of text in a single font.

Drawing text is a two step process:

1. Create a *Text* instance, specifying the text to display and the font to use; and optionally the position, color and alignment of the text.

2. Each frame, call *Text.draw* to draw the text to the window.

The following is a reprint of the "Hello, World" example from the first chapter:

```
from pyglet import window
from pyglet import font

win = window.Window()

ft = font.load('Arial', 36)
text = font.Text(ft, 'Hello, World!')

while not win.has_exit:
    win.dispatch_events()

    win.clear()
    text.draw()
    win.flip()
```

Position and alignment

You can specify the location in the window of the text during construction:

```
text = font.Text(ft, 'Hello, World!', x=25, y=80)
```

or change this at any time (for example, to animate some text moving across the screen):

```
text.x += 10
text.y += 1
```

By default the position you give determines the left edge of the text, and the baseline of the first line of text. The *halign* and *valign* properties allow you to center or right-align text, and position relative to the top, center or bottom of the text. For example, in the following example the text is centered within the window:

```
text = font.Text(ft, 'Hello, World!',
                 x=win.width / 2,
                 y=win.height / 2,
                 halign=font.Text.CENTER,
                 valign=font.Text.CENTER)
```

You may find it useful to be able to align various assets around the text. To do this, you need to know how much space it is taking. The *width* and *height* attributes can be queried; these give the total dimensions of the text when it is laid out.

Text color

The color of the text is white by default, but can be changed with the *color* property:

```
text.color = (1, 0, 0, 1) # Red
```

Colors are specified as if passed to *glColor4f*. They are a tuple of the red, green, blue and alpha color components, in that order. The range of each component is a float between 0 and 1.

For example, to draw semi-translucent dark blue text:

```
text.color = (0, 0.5, 0, 0.5)
```

Word wrapping

Text can be made to fit within a specified width. When the *width* property is not `None`, text will be split into separate lines along word boundaries so that the width of the text does not exceed the limit. For example, to word-wrap a large paragraph into the whole window:

```
text = font.Text(ft, long_text,
                 x=0,
                 y=win.height,
                 valign=font.Text.TOP)
text.width = win.width
```

If you add the `text.width = win.width` line to an *on_resize* event handler for the window, you can ensure the text reflows as the window is resized.

Limitations

The *Text* class is suitable only for text that can be displayed with a single font, size and color. It does not support any markup, nor does it handle bidirectional text.

For more sophisticated text rendering, you may want to consider the `layout` module located in the `contrib` directory of the SVN source tree. The `layout` module allows text to be marked-up with XML or HTML, and then styled with CSS. This module is currently in development and is not yet supported.

The *GlyphString* class gives lower-level access to some of the features provided by *Text*, which could be useful in implementing your own layout engine.

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; for example, when drawing or measuring a *Text*.

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 closing the only window, then creating a new one and reusing the same font. In this case, the original object space was destroyed with the first window's context, so could not be shared with the second window's context.
- 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:

```
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.

For complete control over how text is rendered, use the *Glyph* objects returned from *Font.get_glyphs* to construct your own vertex arrays.

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.

Some simple methods are provided for drawing images to windows; these will usually be sufficient for writing simple sprite-based games. For more advanced imaging you will need some knowledge of OpenGL.

Loading an image

Images are loaded using the *pyglet.image.load* function:

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

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 = 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`	Wrapper interface around PIL Image class.
`pyglet.image.codecs.png`	`PNGImageDecoder`	PNG decoder written in pure Python.

Module	Class	Description
`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 = 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	
`.png`	Portable Network Graphic	X	X	X
`.pnm`	PBM Portable Any Map Graphic Bitmap			X

Extension	Description	Windows XP	Mac OS X	Linux [9]
`.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 *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.

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.

Displaying images

Use the *blit* method to draw an image to the current window:

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

You can use the *blit* method to draw any image, including an image region. The *x* and *y* coordinates locate where to draw the bottom-left corner of the image.

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.

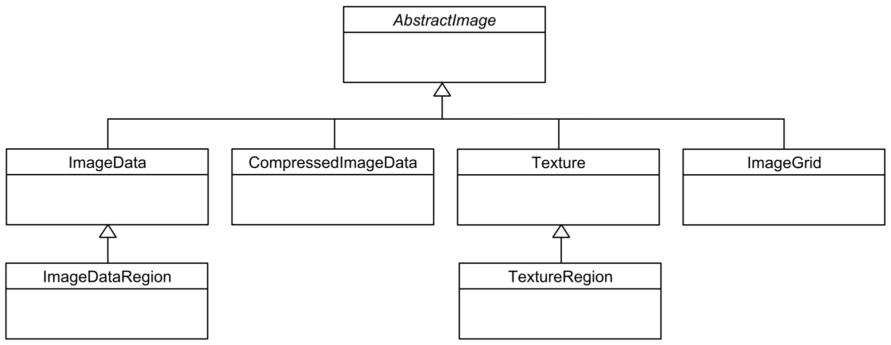
Images with an alpha channel can be blended with the existing framebuffer. To do this you need to supply OpenGL with a blend equation. The following code fragment implements the most common form of alpha blending, however other techniques are also possible:

```
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.

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 *texture* and *image_data* properties defined on *AbstractImage*. For example, to load an image and work with it as an OpenGL texture:

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

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

Original class	**`.texture`**	**`.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]

Original class	**`.texture`**	**`.image_data`**
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.texture.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 by accessing the *image_data* property:

```
kitten = image.load('kitten.png').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, set the *pitch* and *format* properties then read *data*. The following example reads tightly packed rows in `RGB` format (the alpha component, if any, will be discarded):

```
kitten = kitten.image_data
```

```
kitten.format = 'RGB'
kitten.pitch = len(kitten.format) * kitten.width
data = kitten.data
```

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, set the format and pitch first, then the data:

```
kitten.format = 'RGB'
kitten.pitch = kitten.width * 3
kitten.data = 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.

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*, access its *texture* property:

```
kitten = image.load('kitten.jpg')
texture = kitten.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.texture
<TextureRegion 235x257>     # The returned texture is a region
>>> rgba.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.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_for_size` method to create a texture with non-power-2 dimensions using the `GL_texture_rectangle_ARB` extension (the *media* package does this for video textures where possible).

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.

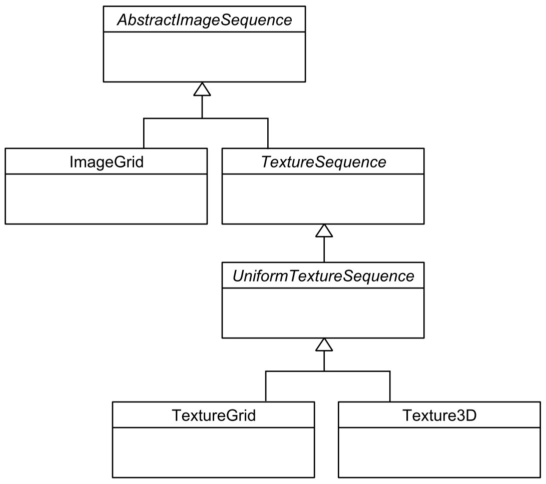
Format	**Internal format**
Any format with 3 components	`GL_RGB`
Any format with 2 components	`GL_LUMINANCE_ALPHA`
`"A"`	`GL_ALPHA`
`"L"`	`GL_LUMINANCE`
`"I"`	`GL_INTENSITY`
Any other format	`GL_RGBA`

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_for_size` class method to create a texture with a specific internal format.

Image sequences

pyglet provides an abstraction for a sequence of images -- for example, an animation consisting of discrete image frames.



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:

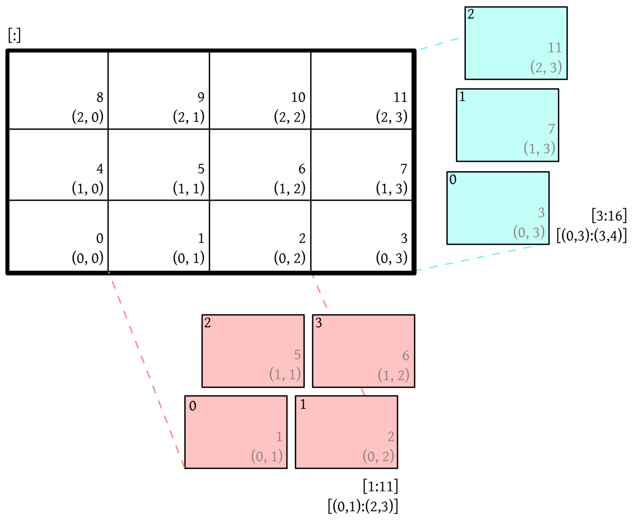
```
from pyglet import image

explosion = image.load('explosion.png')
explosion_seq = 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:

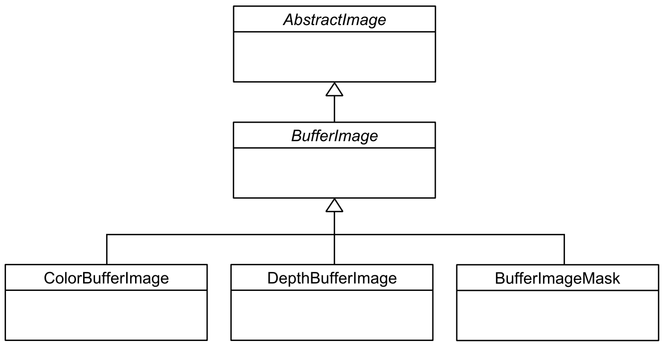
```
explosion_3d = image.Texture3D.create_for_image_grid(explosion_seq)
```

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.

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.

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:

```
from pyglet import image

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:

```
import pyglet
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:

```
import pyglet
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:

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

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 = 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 = media.StaticSource(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:

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

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

pyglet is not multi-threaded. In order to ensure the audio buffers remain full, you must call *media.dispatch_events* periodically. For example, it is common to do this in your main run loop:

```
while True:
    media.dispatch_events()
```

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:

```
from pyglet import media
player = 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

Property	**Description**
	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.

While a player is playing, it is important that your application periodically calls its *dispatch_events* method. This is used to update the video texture and to ensure the audio buffers remain full. Typically this call is done in every iteration of your main loop:

```
while True:
    player.dispatch_events()
```

Note that this *dispatch_events* method is unrelated to the *media.dispatch_events* function, which provides the same functionality for *ManagedPlayer* instances.

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, its *texture* property gives the current video image. This can be used to display the current video image syncronised with the audio track, for example:

```
while True:
    # Fill audio buffers and update the video texture
    player.dispatch_events()

    # Draw the video image to the current window.
    player.texture.blit(0, 0)
```

Because *Player* uses OpenGL to create video textures, you must ensure an OpenGL context is created before queueing a video source onto a player. You normally do this by creating a *Window*.

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.

Advanced topics

Environment settings

Options in the *pyglet.options* dictionary can have defaults set through the operating system's environment variable. The following table shows which environment variable is used for each option:

Environment variable	***pyglet.options* key**	**Type**	**Default value**
`PYGLET_AUDIO`	`audio`	List of strings	`directsound,opeanl,alsa,silen`
`PYGLET_DEBUG_GL`	`debug_gl`	Boolean	`1` [12]

[12]Defaults to `1` unless Python is run with `-O` or from a frozen executable.