

# The OpenGL interface

pyglet provides a direct interface to OpenGL. 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 CPU. You can access this interface directly; using it is much like using OpenGL from C.

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

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

## Using OpenGL

Documentation for OpenGL is provided at the [OpenGL website](#) and (more comprehensively) in the [OpenGL Programming SDK](#).

Importing the package gives access to OpenGL 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 code sets the GL clear color and enables depth testing and face culling:

```
from pyglet.gl import *

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

glClearColor(1, 1, 1, 1)
glEnable(GL_DEPTH_TEST)
glEnable(GL_CULL_FACE)
```



 latest ▼

Some OpenGL functions require an array of data. These arrays must be constructed as `ctypes` arrays of the correct type. The following example shows how to construct arrays using OpenGL types:

```
from pyglet.gl import *

# Create a new array type of length 32:
array32f = GLfloat * 32

# Create an instance of this array with initial data:
array_instance = array32f(*data)

# More commonly, combine these steps:
array_instance = (GLfloat * 32)(*data)
```

Similar array constructions can be used to create data for other OpenGL objects.

## 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`. pyglet Windows have a `projection` property that can be set with a 4x4 projection matrix. See [Matrix and Vector Math](#) for more information on creating matrixes. The default `on_resize` handler is defined as:

```
@window.event
def on_resize(width, height):
    glViewport(0, 0, *window.get_framebuffer_size())
    window.projection = Mat4.orthogonal_projection(0, width, 0, height, -255, 255)
```

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

```
@window.event
def on_resize(width, height):
    glViewport(0, 0, *window.get_framebuffer_size())
    window.projection = Mat4.perspective_projection(window.aspect_ratio, z_near=0.1,
    z_far=255)
    return pyglet.event.EVENT_HANDLED
```

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

# Error checking

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

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

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

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

from pyglet.gl import *
```

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

## Using extension functions

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

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

You can also easily check the version of OpenGL:

```
if pyglet.gl.gl_info.have_version(4, 6):
    # We can assume all OpenGL 4.6 functions are implemented.
```

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

# Using multiple windows

pyglet allows you to create and display any number of windows simultaneously. Each will be created with its own OpenGL context, however all contexts will share the same texture objects, display lists, shader programs, and so on, by default <sup>1</sup>. Each context has its own state and framebuffers.

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

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

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


## AGL, GLX and WGL

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

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

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

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

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

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

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

Develop and launch modern apps with MongoDB Atlas, a resilient data platform.

*Ads by EthicalAds*