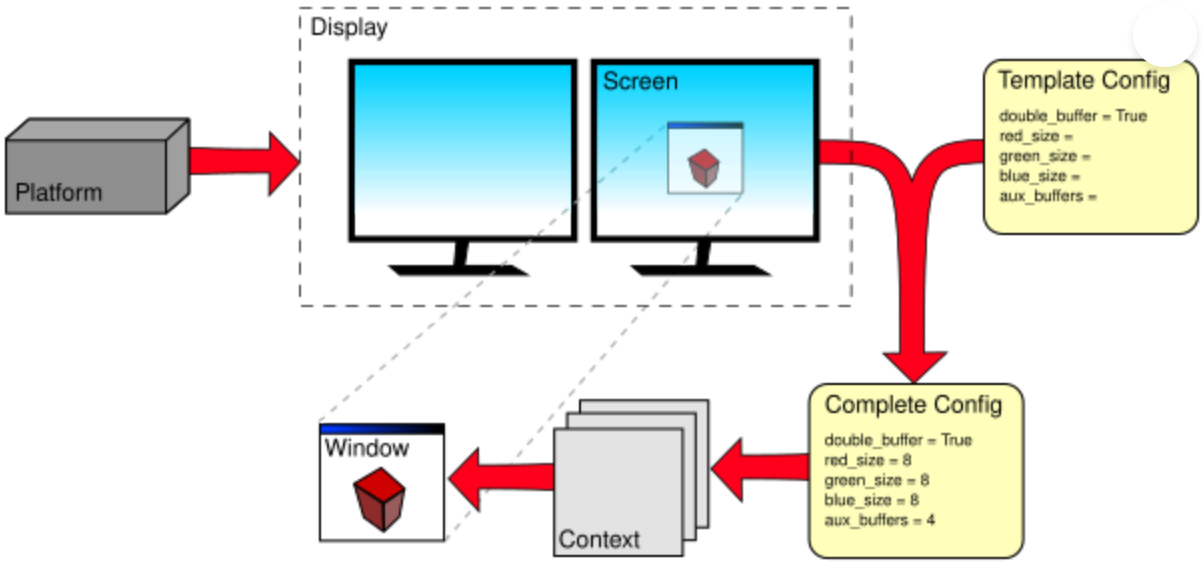
Creating an OpenGL context

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 abstract Display 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 som properties:

;

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

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

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

Displays

The system may actually support several different sets of configs, depending on which display device is being used. For example, a computer with two video cards may 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 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).

The `pyglet.display` module provides access to the display(s). Use the `get_display()` function to get the default display:

```
>>> display = pyglet.display.get_display()
```

Note

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

```
>>> display = pyglet.display.Display(name=':1')
```

If you have multiple physical screens and you're using Xinerama, see Screens to select the desired screen as you would for Windows and Mac OS X. Otherwise, you can specify the screen number via the `x_screen` argument:

```
>>> display = pyglet.display.Display(name=':1', x_screen=1)
```

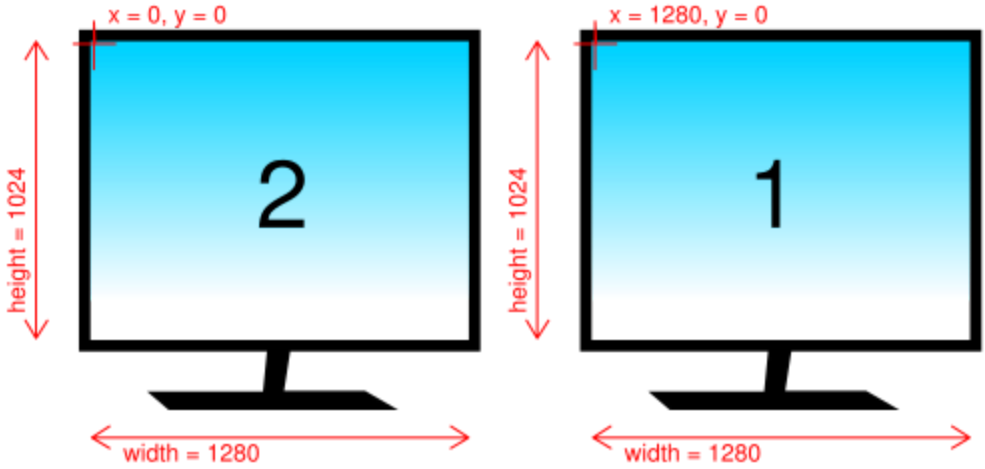
Screens

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.



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 (MSAA), 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 (MSAA) 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. `samples=4` is still widely supported and fairly performant even on Intel HD and AMD Vega. Most modern GPUs support 2×, 4×, 8×, and 16× MSAA samples with fairly high performance.

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

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

If you wish to work with OpenGL directly, you can request a higher level context. This is required if you wish to work with the modern OpenGL programmable pipeline. Please note, however, that pyglet currently uses legacy OpenGL functionality for many of its internal modules (such as the text, graphics, and sprite modules). Requesting a higher version context will currently prevent usage of these modules.

`major_version`

This will be either 3 or 4, for an OpenGL 3.x or 4.x context.

`minor_version`

The requested minor version of the context. In some cases, the OpenGL driver may return a higher version than requested.

`forward_compatible`

Setting this to *True* will ask the driver to exclude legacy OpenGL features from the context. Khronos does not recommend this option.

Note

To request a higher higher version OpenGL context on Mac OSX, it is necessary to disable the pyglet shadow context. To do this, set the pyglet option `pyglet.options['shadow_window']` to `False` *before* creating a Window, or importing `pyglet.window`.

The default configuration

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

Attribute	Value
double_buffer	True
depth_size	24

Simple context configuration

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

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

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

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

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

```
display = pyglet.display.get_display()
screen = display.get_default_screen()

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

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

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

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

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

Selecting the best configuration

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

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

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

```
display = pyglet.display.get_display()
screen = display.get_default_screen()

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

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

Sharing objects between contexts

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

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

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

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

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

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

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

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

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

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

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

```
context = pyglet.gl.current_context
object_space = context.object_space
object_space.my_game_objects_loaded = True
```

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