The application event loop

In order to let pyglet process operating system events such as mouse and keyboard events, applications need to enter an application event loop. The event loop watches for new events, dispatches those events, and sleeps until something else needs to be done. It also handles any functions that you have scheduled on the clock (see Calling functions periodically). pyglet ensures that the loop iterates only as often as necessary to fulfill all scheduled functions and user input. It is well tuned for performance and low power usage on Windows, Linux and macOS.

After creating Windows and attaching event handlers, most applications need only call:

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

The `run()` function does not return until all open windows have been closed, or until `pyglet.app.exit()` is called.

Once you have entered the event loop it dispatches window events (such as for keyboard input or mouse movement), events from Controllers or Joysticks, and any other events as they occur. By default, the application event loop will also refresh all Windows and dispatch the `on_draw()` event at a rate of 60Hz (60 times per second). You can customize this by passing the desired interval in seconds to `run()`:

```
pyglet.app.run(1/30)    # 30Hz
# or
pyglet.app.run(1/120)   # 120Hz
# or for benchmarking, redraw as fast as possible:
pyglet.app.run(0)
```

Passing *None* to `run()` is a special case. It will enter the event loop as usual, but it will not dispatch the Window events. This can be desired if you wish to have different refresh rates for different Windows, or even change the refresh rate while the application is runnin

Customising the event loop

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

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

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

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

Event loop events

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

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

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

event_loop.run()
```

Overriding the default idle policy

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

latest

The default implementation has the following algorithm:

1. Call `pyglet.clock.tick()` with `poll=True` to call any scheduled functions.
2. Return the value of `pyglet.clock.get_sleep_time()`.

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

Creating a Custom Event Loop

Many windowing toolkits requie the application developer to write their own event loop. This is also possible in pyglet, but is usually just an inconvenience compared to `pyglet.app.run()`. It can be necessary in some situations, such as when combining pyglet with other toolkits, but is strongly discouraged for the following reasons:

- Keeping track of delta times between frames, and maintaining a stable frame rate can be challenging. It is difficult to write a manual event loop that does not waste CPU cycles and is still responsive to user input.
- The `EventLoop` class provides plenty of hooks for most toolkits to be integrated without needing to resort to a manual event loop.
- Because `EventLoop` is tuned for specific operating systems, it is more responsive to user events, and continues calling clock functions while windows are being resized, and (on macOS) the menu bar is being tracked.

With that out of the way, a manual event loop usually has the following form:

```
while True:
    pyglet.clock.tick()
    pyglet.app.platform_event_loop.step(timeout)

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

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

The `dispatch_events()` method checks the window's operating system event queue for user input and dispatches any events found. The method does not wait for input – if t
events pending, control is returned to the program immediately.

latest

The `dispatch_event('on_draw')()` method is optional if you are catching this Window event. If you are not using this event, your draw calls (*Batch.draw()*) should go here instead.