Controller & Joystick input

The `input` module allows you to accept input from USB or Bluetooth human interface devices (HID). High-level classes are provided for working with game controllers, joysticks, and the Apple Remote, with named and normalized inputs. Basic support is also provided for Drawing Tablets, such as those made by Wacom.

The game controller abstraction is most suitable for modern dual-analog stick controllers, such as those from video game consoles. The joystick abstraction is more generalized, and suits devices with an arbitrary number of buttons, absolute or relative axis, and hats. This includes devices like flight sticks, steering wheels, and just about anything else with digital and/or analog inputs. For most types of games, the game controller abstraction is recommended.

For advanced use cases, it is also possible to access the low-level input devices directly. This can be useful if you need direct accesss to the raw inputs, without normalization. For most application and games this is not required.

The `input` module provides several methods for querying devices, and a ControllerManager class to support hot-plugging of Controllers:

```
# get a list of all low-level input devices:
devices = pyglet.input.get_devices()

# get a list of all controllers:
controllers = pyglet.input.get_controllers()

# get a list of all joysticks:
joysticks = pyglet.input.get_joysticks()

# get a list of tablets:
tablets = pyglet.input.get_tablets()

# get an Apple Remote, if available:
remote = pyglet.input.get_apple_remote()

# create a ControllerManager instance:
controller_manager = pyglet.input.ControllerManager()
```

Using Controllers

Controllers have a strictly defined set of inputs that mimic the layout of modern dual-analog stick video game console Controllers. This includes two analog sticks, analog triggers, a directional pad (dpad), face and shoulder buttons, and start/back/guide and stick press buttons. Many controllers also include the ability to play rumble effects (vibration). The following platform interfaces are used for Controller support:

platform	interface	notes
Linux	evdev	
Windows	DirectInput & Xinput	rumble not implemented on DirectInput
MacOSX	IOKit	rumble not yet implemented

Before using a controller, you must find it and open it. You can either list and open Controllers manually, or use a ControllerManager. A ControllerManager provides useful events for easily handling hot-plugging of Controllers, which is described in a following section. First, however, lets look at how to do this manually. To get a list of all controllers currently connected to your computer, call pyglet.input.get_controllers():

```
controllers = pyglet.input.get_controllers()
```

Then choose a controller from the list and call *Controller.open()* to open it:

```
if controllers:
    controller = controllers[0]
    controller.open()
```

Once opened, you you can start receiving data from the the inputs. A variety of analog and digital `Control` types are defined, which are automatically normalized to consistent ranges. The following analog controls are available:

name	type	range
leftx	float	-1~1
lefty	float	-1~1
rightx	float	-1~1
righty	float	-1~1
dpadx	float	-1~1
dpady	float	-1~1
lefttrigger	float	0~1
righttrigger	float	0~1

The following digital controls are available:

Name	notes
a	the “south” face button
b	the “east” face button
x	the “west” face button
y	the “north” face button
leftshoulder	
rightshoulder	
start	called “options” on some controllers
back	called “select” or “share” on some controllers
guide	usually in the center, with a company logo
leftstick	pressing in on the left analog stick
rightstick	pressing in on the right analog stick

These values can be read in two ways. First, you can just query them manually in your game loop. All control names listed above are properties on the controller instance:

```
# controller_instance.a       (boolean)
# controller_instance.leftx   (float)

if controller_instance.a == True:
    # do something
```

latest

Alternatively, since controllers are a subclass of `EventDispatcher`, events will be dispatched when any of the values change. This is usually the recommended way to handle input, since it reduces the chance of “missed” button presses due to slow polling. The different controls are grouped into the following event types:

Event	Arguments	types
on_button_press	controller, button_name	`Controller`, *str*
on_button_release	controller, button_name	`Controller`, *str*
on_stick_motion	controller, stick_name, vector	`Controller`, *str*, `Vec2`
on_dpad_motion	controller, left, right, up, down	`Controller`, `Vec2`
on_trigger_motion	controller, trigger_name, value	`Controller`, *str*, *float*

Analog (and Dpad) events can be handled like this:

```
@controller.event
def on_stick_motion(controller, name, vector):

    if name == "leftstick":
        # Do something with the 2D vector
    elif name == "rightstick":
        # Do something with the 2D vector

@controller.event
def on_trigger_motion(controller, name, value):

    if name == "lefttrigger":
        # Do something with the value
    elif name == "righttrigger":
        # Do something with the value

@controller.event
def on_dpad_motion(controller, vector):
    # Do something with the 2D vector
```

Digital events can be handled like this:

latest

```
@controller.event
def on_button_press(controller, button_name):
    if button_name == 'a':
        # start firing
    elif button_name == 'b':
        # do something else


@controller.event
def on_button_release(controller, button_name):
    if button_name == 'a':
        # stop firing
    elif button_name == 'b':
        # do something else
```

Rumble

Many controllers also support playing rumble (vibration) effects. There are both strong and weak effects, which can be played independently:

```
controller.rumble_play_weak(strength, duration=0.5)
controller.rumble_play_strong(strength, duration=0.5)
```

The *strength* parameter should be on a scale of 0-1. Values outside of this range will be clamped. The optional *duration* parameter is in seconds. The maximum duration can vary from platform to platform, but is usually at least 5 seconds. If you play another effect while an existing effect is still playing, it will replace it. You can also stop playback of a rumble effect at any time:

```
controller.rumble_stop_weak()
controller.rumble_stop_strong()
```

ControllerManager

To simplify hot-plugging of Controllers, the `ControllerManager` class is available. This class has a *get_controllers()* method to be used in place of *pyglet.input.get_controllers()*. There are also *on_connect* and *on_disconnect* events, which dispatch a Controller instance whenever one is connected or disconnected. First lets review the basic functionality.

To use a ControllerManager, first create an instance:

```
manager = pyglet.input.ControllerManager()
```

You can then query the currently connected controllers from this instance. (An empty list is returned if no controllers are detected):

```
controllers = manager.get_controllers()
```

Choose a controller from the list and call *Controller.open()* to open it:

```
if controllers:
    controller = controllers[0]
    controller.open()
```

To handle controller connections, attach handlers to the following methods:

```
@manager.event
def on_connect(controller):
    print(f"Connected:  {controller}")

@manager.event
def on_disconnect(controller):
    print(f"Disconnected:  {controller}")
```

Those are the basics, and provide the building blocks necessary to implement hot-plugging of Controllers in your game. For an example of bringing these concepts together, have a look at `examples/input/controller.py` in the repository.

Note

If you are using a ControllerManager, then you should not use *pyglet.input.get_controllers()* directly. The results are undefined. Use *ControllerManager.get_controllers()* instead.

Using Joysticks

Before using a joystick, you must find it and open it. To get a list of all joystick devices currently connected to your computer, call `pyglet.input.get_joysticks()`:

```
joysticks = pyglet.input.get_joysticks()
```

Then choose a joystick from the list and call *Joystick.open* to open the device:

```
if joysticks:
    joystick = joysticks[0]
    joystick.open()
```

The current position of the joystick is recorded in its 'x' and 'y' attributes, both of which are normalized to values within the range of -1 to 1. For the x-axis, *x* = -1 means the joystick is pushed all the way to the left and *x* = 1 means the joystick is pushed to the right. For the y-axis, a value of *y* = -1 means that the joystick is pushed up and a value of *y* = 1 means that the joystick is pushed down. If other axis exist, they will be labeled *z*, *rx*, *ry*, or *rz*.

The state of the joystick buttons is contained in the *buttons* attribute as a list of boolean values. A True value indicates that the corresponding button is being pressed. While buttons may be labeled A, B, X, or Y on the physical joystick, they are simply referred to by their index when accessing the *buttons* list. There is no easy way to know which button index corresponds to which physical button on the device without testing the particular joystick, so it is a good idea to let users change button assignments.

Each open joystick dispatches events when the joystick changes state. For buttons, there is the `on_joybutton_press()` event which is sent whenever any of the joystick's buttons are pressed:

```
def on_joybutton_press(joystick, button):
    pass
```

and the `on_joybutton_release()` event which is sent whenever any of the joystick's buttons are released:

```
def on_joybutton_release(joystick, button):
    pass
```

The `Joystick` parameter is the `Joystick` instance whose buttons changed state (useful if you have multiple joysticks connected). The *button* parameter signifies which button changed and is simply an integer value, the index of the corresponding button in the *buttons* list.

For most games, it is probably best to examine the current position of the joystic latest using the *x* and *y* attributes. However if you want to receive notifications whenever these values change you should handle the `on_joyaxis_motion()` event:

```
def on_joyaxis_motion(joystick, axis, value):
    pass
```

The `Joystick` parameter again tells you which joystick device changed. The *axis* parameter is string such as “x”, “y”, or “rx” telling you which axis changed value. And *value* gives the current normalized value of the axis, ranging between -1 and 1.

If the joystick has a hat switch, you may examine its current value by looking at the *hat_x* and *hat_y* attributes. For both, the values are either -1, 0, or 1. Note that *hat_y* will output 1 in the up position and -1 in the down position, which is the opposite of the y-axis control.

To be notified when the hat switch changes value, handle the `on_joyhat_motion()` event:

```
def on_joyhat_motion(joystick, hat_x, hat_y):
    pass
```

The *hat_x* and *hat_y* parameters give the same values as the joystick’s *hat_x* and *hat_y* attributes.

A good way to use the joystick event handlers might be to define them within a controller class and then call:

```
joystick.push_handlers(my_controller)
```

Please note that you need a running application event loop for the joystick button an axis values to be properly updated. See the The application event loop section for more details on how to start an event loop.

Using the Apple Remote

The Apple Remote is a small infrared remote originally distributed with the iMac. The remote has six buttons, which are accessed with the names *left*, *right*, *up*, *down*, *menu*, and *select*. Additionally when certain buttons are held down, they act as virtual buttons. These are named *left_hold*, *right_hold*, *menu_hold*, and *select_hold*.

To use the remote, first call `get_apple_remote()`:

```
remote = pyglet.input.get_apple_remote()
```

Then open it:

```
if remote:
    remote.open(window, exclusive=True)
```

The remote is opened in exclusive mode so that while we are using the remote in our program, pressing the buttons does not activate Front Row, or change the volume, etc. on the computer.

The following event handlers tell you when a button on the remote has been either pressed or released:

```
def on_button_press(button):
    pass

def on_button_release(button):
    pass
```

The *button* parameter indicates which button changed and is a string equal to one of the ten button names defined above: “up”, “down”, “left”, “left_hold”, “right”, “right_hold”, “select”, “select_hold”, “menu”, or “menu_hold”.

To use the remote, you may define code for the event handlers in some controller class and then call:

```
remote.push_handlers(my_controller)
```

Low-level Devices

It’s usually easier to use the high-level interfaces but, for specialized hardware, the low-level device can be accessed directly. You can query the list of all devices, and check the *name* attribute to find the correct device:

```
for device in pyglet.input.get_devices():
    print(device.name)
```

After identifying the Device you wish to use, you must first open it:

```
device.open()
```

Devices contain a list of `Control` objects. There are three types of controls: `Button`, `AbsoluteAxis`, and `RelativeAxis`. For helping identify individual controls, each control has at least a *name*, and optionally a *raw_name* attribute. Control values can by queried at any time by checking the *Control.value* property. In addition, every control is also a subclass of `EventDispatcher`, so you can add handlers to receive changes as well. All Controls dispatch the *on_change* event. Buttons also dispatch *on_press* and *on_release* events.:

```
# All controls:

@control.event
def on_change(value):
    print("value:", value)

# Buttons:

@control.event
def on_press():
    print("button pressed")

@control.event
def on_release():
    print("button release")
```