Pygame Tutorials

Input Handling

by Pete Shinners

pete@shinners.org

Revision 1.0, March 28th, 2004

Handling Input Events in Pygame is not difficult, but there are several techniques that can be used. You will want to use the correct method for the correct job. The two main techniques are *Event Handling* and *State Checking*. This tutorial will look at each of these, as well as show how to handle the keyboard, mouse, and joysticks.

State Checking

State Checking simply means calling a function to check the current position or value of an input device. This is usually the simplest way a program can figure out what is going on. Your program calls one of the state functions and knows immediately what the user is doing. The state is connected directly to the input device, so you know exactly what is going on.

Each Input device has an object that provides several methods to check the current state.

- **pygame.mouse**: This is a module that contains functions for accessing the mouse. This module can also change the system cursor graphics, control visibility, and actually reposition the mouse.
    - **get_pos()**: Returns the X,Y pair of the mouse position. Relative to the top left corner of the window.
    - **get_rel()**: Returns the X,Y pair of mouse movement since the last time get_rel() was called. This is the only function of its kind, that is based on previous calls.
    - **get_pressed()**: Returns three values representing the pressed state of three mouse buttons. 0 values mean not pressed, 1 values mean pressed.
- **pygame.key**: This is a module that contains functions for accessing the keyboard. This module can control keyboard repeat rates and translate key id values into english names.
    - **get_pressed()**: Returns a tuple representing the pressed state for every key on the keyboard. Use key id values to index the state of an individual button.
    - **get_mods()**: Returns an integer representing the pressed state of all the modifier keys (Alt, Control, Shift, Locks, etc). The integer is a bitwise array of each key, using the modifier key id values. To test if a key or combination of keys is pressed, use python bit masking operators.
        - examples
            - pygame.key.get_mods() & KMOD_SHIFT
                - true if either shift key is pressed
            - pygame.key.get_mods() & KMOD_RCTRL
                - true if right control key is pressed
- **Joystick**: An object created from the function pygame.joy.Joystick(id). Each joystick ojbect has an init() method that must be called before any of these state methods will work.
    - **get_axis(axisnum)**: Returns the position of a specific axis on the joystick. Axis 0 is usually left/right and Axis 1 is up/down. Other axis can represent throttle or twists on the joystick device. The position is a value between -1 and 1, with 0 being the center.

- **get_ball(ballnum)**: Report the X,Y position of a trackball located on the joystick device. The balls work similar to a regular mouse position.
- **get_button(buttonnum)**: Report the pressed state of a button on the joystick device. 0 represents not pressed, and 1 represents pressed.
- **get_hat(hatnum)**: Get the current position of a hat control. Hats are like miniature joysticks on top of joystick devices that can report simple directions. Returns an X,Y pair of the position of the hat. Values from -1 to 1, with 0 being the center.

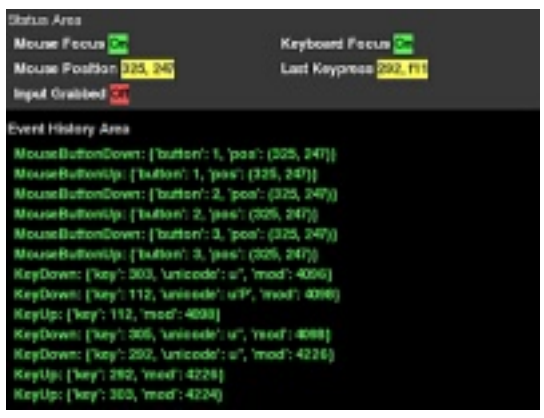
But there are several problems writing a game with *State Checking* as the only input control. First, you have no way to know the order of actions. When calling button press checking function, there could be 2 or 3 buttons all pressed at once, you have no way to determine the order of pressing. The second problem is you can completely "miss" button pushes. If the user clicks very fast, between your calls to check the button state, you will never know anything changed. These problems are solved with event handling.

Another problem to avoid, if your game calls these checking functions multiple times computing the current frame, the value can switch partway through computing your frame. You may want to save the state into a variable and always check with that variable.

Event Handling

The preferred way of dealing with input is *Event Handling*. Event handling is commonly used in graphical interface programs. The system keeps a list of things that have happened, and your program can process this, usually once per frame. This list of events is known as the *Event Queue*, and it allows you to know the order of everything that happened since your last time checking the queue.

The pygame examples includes an excellent event inspector program named, "eventlist.py"



Pygame has several methods for managing the queue, all in the *pygame.event* module. Be aware that Pygame's queue is lower level than what you might be used to in bigger GUI frameworks. It is up to your game to manage everything about the queue. The main function you will use is *pygame.event.get()*, which removes the events in the queue and returns them inside a list.

The events are a single Event object type. Each event has a specific type ID attribute named "id". It also has several other named attributes specific to that type. This outline shows the input event ID's and the attributes they carry.

- Mouse Events
  - **MOUSEMOTION**: occur frequently as the mouse is moving
    - **pos**: current X,Y position of the mouse, relative to the topleft window corner
    - **rel**: amount of X,Y relative motion in this mouse event
    - **buttons**: three values representing the state of each mouse button during this move
  - **MOUSEBUTTONUP**: occur when any mouse button is pressed
  - **MOUSEBUTTONDOWN**: occur when any mouse button is released
    - **button**: number value representing the mouse button pressed or released
    - **pos**: X,Y mouse position when the button was pressed or released
- Keyboard Events

    - **KEYDOWN**: occur when any keyboard button is released
    - **KEYUP**: occur when any keyboard button is pressed
        - **key**: key id of the button that was pressed or released
        - **mod**: state of keyboard modifiers when the button was pressed or released
        - **unicode**: represents the system translated keypress into a unicode character string, only on KEYDOWN events
- Joystick Events
    - **JOYAXISMOTION**: occur when joystick axis changes
        - **joy**: joystick id of the event
        - **axis**: axis id of the event
        - **pos**: new position of the axis, -1 to 1 with 0 the center
    - **JOYBALLMOTION**: occur when joystick ball rotates
        - **joy**: joystick id of the event
        - **ball**: ball id of the event
        - **rel**: X,Y movement of the ball
    - **JOYHATMOTION**: occur when joystick hat changes
        - **joy**: joystick id of the event
        - **hat**: hat id of the event
        - **value**: X,Y position of the hat, -1 to 1 with 0 the center
    - **JOYBUTTONUP**: occur when joystick button pressed
    - **JOYBUTTONDOWN**: occur when joystick button released
        - **joy**: joystick id of the event
        - **button**: joystick button pressed or released

When you want to move objects while certain buttons are held down, you should check the event queue, and set the state of interesting buttons to global variables. Then your code can check the state of those variables.

Keyboard Control

The keyboard is probably the simplest input device. Keys are represented by their key id value. The only real control you have over the keyboard is setting repeat rates. By default, pygame sends a single KEYDOWN and KEYUP event for every keypress. You can enable key repeating with *pygame.key.set_repeat()*. The defaults are usually fine, but you can fine tune the repeat behavior. When keys are repeating, you will receive multiple KEYDOWN events for as long as the key is held, and a final KEYUP when it is released.

One other common need is getting text entry from the keyboard. This involved proper capitalization with the shift keys, as well as special input handling on international keyboards. This is a complex task to do by yourself. Fortunately, Pygame already provides these translations with every KEYDOWN event. The unicode attribute is the "system translated" representation of the key. For special control keys it can often be an empty string.

One last keyboard helpful tip. Remember that the keycode for the main ENTER key is K_RETURN. The keycode for the keypad enter key is K_ENTER. See the full list of keyboard keycode's here, in the reference documentation.

Mouse Control

Mouse input is fairly straightforward. There are a couple extra features you may want to be aware of. The mouse wheel is emulated in pygame through buttons 4 and 5. The only way to receive these events is from the MOUSEBUTTONDOWN events. This also means Pygame doesn't handle extended mouse buttons beyend the regular 3.

Pygame can enable a special "virtual infinite area", best used for fullscreen games. Typical behavior is the mouse is trapped to the screen edges in fullscreen mode. If you only care about relative mouse movement, this can present a problem. Even if the mouse cursor is hidden, you will stop receiving relative motion past the edges of the screen. To enable the "virtual infinite area" you must set the mouse cursor to invisible, then also grab the input focus with *pygame.event.set_grab(1)*.

Managing the Event Queue

Your program must deal with the input event queue as it runs. Even if your program doesn't use events, there are a few things you'll need to watch for. The first thing to be aware of is that the queue does not have infinite size. Once the queue fills up, new events can no longer be created. The queue is plenty large to hold events for a single frame, but you if you ignore it for too long it will fill up. Especially with MOUSEMOTION events that get created frequently.

Another thing to be aware of is that your application needs to do some coordination with the graphic environment it lives in. This is especially important for windowed games, but still necessary for fullscreen games as well. When you call the Pygame event queue functions, Pygame will take a moment to cooperate with the graphics environment, and potentially even create new events on the queue itself. These are events like QUIT, VIDEORESIZE, and others that let your program know what is going on. This makes it important that your application calls at least one pygame.event function, usually once per frame. If you do not care at all about the Pygame queue, you can call *pygame.event.pump()*, which will allow Pygame to do its necessary processing. You may also consider *pygame.event.clear()* to do the same thing, but keep the queue empty.

Some programs and games reach a point where they are waiting on user input. This is common in image viewers, simple paint programs, or even turn based games. If you are creating this type of program and would like to be extremely cooperative with other running applications, you can use *pygame.event.wait()*. If no events are available, your program will be put to sleep by the operating system, until some events becomes available. This means your program will take 0% cpu time while it is waiting for the users actions.

Pygame's event queue also has several ways to deal with events of a specific type. First you can call *pygame.event.set_blocked(events)*, which prevents certain types of events from even entering the queue. You can also pass a list of interesting event types to *pygame.event.get()*, and you will only receive events of the type you asked for. Be careful passing this mask into pygame.event.get. All other event types will remain on the queue, and will eventually fill it up if you never ask for them.

Custom Events

Pygame reserves a set of event ID values for custom use in your program. These are values between USEREVENT and NUMEVENTS. It is up to your program to coordinate how these events are used. Certain pygame functions will require an event ID for creating their own events. For example, you can have Sound objects create an event of any type when they are finished playing.

To create your own events you first create an Event object, and pass it to *pygame.event.post()*. The Event

objects are easy to create, you simply pass an event type ID and a list of keyword named arguments. Examples may make this easier.

```
PLAYERDEAD = USEREVENT+2
deadevent = pygame.event.Event(PLAYERDEAD, player=1, score=game.player1.score)
pygame.event.post(deadevent)
```

It is also simple to set up custom timer events, that are added to the queue for every time interval. You need to call *pygame.time.set_timer(USEREVENT, delay)*. The delay is in milliseconds, and the given event type will appear on the event queue after each amount of time in the delay argument.