## Pygame tutorial #4: more on events

I have received loads of requests for this, but $LIFE took a serious down-turn for a while. Oh well, let’s not waste time with that. On with the show!

**The trouble with poll**

So far, we have been using pygame.event.poll() to get event information. I have already hinted that there are alternatives. But before we do that let us try to see what the issue at hand is.

To begin with run any of the pygame programs that we have written as a part of this tutorial series so far, apart from the very first example from tutorial #1, which didn’t even have an event loop. Before you start the program start up some utility that lets you see the CPU usage. I personally prefer top, but you could use one of those Gnome or KDE thingies as well. If you are under Windows, the Task Manager should do the job.

Notice how sharply the CPU usage shoots up when your program starts. A part of that is inevitable, but you should be able to write pygame applications that behave far better than that. On my system, even the simplest program with just an empty window averages over 80% CPU usage.

Why is that? Well, we can try to add a small print statement to one of our earliest examples and see what results we get.

 1 #! /usr/bin/env python
 2
 3 import pygame
 4
 5 screen = pygame.display.set\_mode((640, 400))
 6 running = 1
 7
 8 **while** running:
 9     event = pygame.event.poll()
10     **if** event.type == pygame.QUIT:
11         running = 0
12     **else**:
13         **print** (event.type)

Now run this and study the output. (Warning: depending on your hardware this can really slow down your machine.) As you can see, the program spits out loads of zeros to the terminal. If you place the mouse over the window and wiggle it a bit, you will see some other number pass by once in a while, but still most of the numbers being printed are zeros.

This is because the poll() method returns an event of type NOEVENT if there is no event in the event queue. This is rather wasteful. We need a better way of writing our event loop. We will try out pygame.event.get() instead. Below is the modified code:

 1 #! /usr/bin/env python
 2
 3 import pygame
 4
 5 screen = pygame.display.set\_mode((640, 480))
 6 running = 1
 7
 8 **while** running:
 9     **for** event **in** pygame.event.get():
10         **if** event.type == pygame.QUIT:
11             running = 0
12         **else**:
13             **print (** event.type)
14

Notice how we now have a little ‘for loop’ instead of just the call to poll(). This loop will go through all the active events on the queue. There is a big difference. Run the program again and notice the output. No zeros! Which means, pygame.event.get() is not generating any NOEVENTs. The only time you will see numbers appearing on the screen is when you move and/or click the mouse over the window. The numbers you see correspond to the event types.

That takes care of part of the problem. However, if you go back to studying your CPU usage you will see that it is probably still high. It may be slightly lower, however. But we can still clearly do something about our CPU usage.

What is happening is that our program is so busy running through the event loop that it puts a lot of demand on the CPU. Imagine if we could get the program to take a pause once in a while, so that the CPU gets a chance to breathe out (and the operating system a chance to deal with other issues). That is exactly what we are going to do with a pygame.time.Clock().

For the time being it may be helpful to understand this as that we have a clock that we use to regularly tell our pygame program to “take a rest”. It will then signal to the rest of the system that, “hey, I’m idle at the moment, so do whatever it is you have to do”.

Creating the Clock() is only the first part, we also need to let it work. This is done inside the event loop. Simply call the tick() method on the Clock object. You can optionally pass in a number that I won’t go into too much because it will spoil one of the exercises. 

The final version of our program then looks as follows:

 1 import pygame
 2
 3 screen = pygame.display.set\_mode((640, 480))
 4 clock = pygame.time.Clock()
 5 running = 1
 6
 7
 8  **while** running:
 9      **for** event **in** pygame.event.get():
10         **if** event.type == pygame.QUIT:
11             running = 0
12
13     clock.tick(20)

Notice that I have used clock.tick(20). In this case there is nothing happening on the screen, so the value I place there is rather irrelevant. But when you add “things” to your screen you will notice that the value you use will make a difference. Without going into too much detail, there is a trade-off between getting your program to run as fast as possible, and your resource usage spiking to the point where the computer becomes very slow.

What I have taught you here obviously means that you have to “unlearn” some of the things I taught you from the beginning. Sorry about that. But then again, it won’t be the last time.

I personally don’t like tutorials that say “do like this, don’t do like that”. I prefer to see things for myself. I hope that I have managed to get you to understand why you should write your programs in a certain way. My long-term goal is not just to let you be able to put together pygame programs, but to know exactly what you are doing as well.

That’s it for now. But this time you won’t have to wait as long for the next tutorial, I promise.

**Exercises**

Just a single exercise this time, lucky you!

1. Rewrite every single application you have written so far as a part of the tutorial series, both the examples I have given as well as the exercise you have solved (because you have been doing the exercises, haven’t you?). In each case, re-write it to use pygame.event.get() instead of pygame.event.poll(). Also include a Clock().

For each program, compare CPU usage and responsiveness between the original version and your new version. Also, experiment with different values in the clock.tick() method call. Try a really large number. Then try a really small number. In each case, look at CPU usage. See how fast or slow the program runs. Based on what you see, what can you say about x in clock.tick(x)? How does a low or a high value of x affect responsiveness and speed of the application as well as CPU usage? Finally try to find a value that keeps your pygame application running as smoothly as possible but without making the rest of your computer too slow.