## Pygame tutorial #5: pixels

Welcome to pygame tutorial #5. This time we’ll look at how to plot individual pixels. We will also look at a keyboard event that may prove useful when we get to the stage of actually being able to write games.

**Setting a pixel**

You can use Surface.set\_at() to plot a single pixel on your surface. You will need to provide the pixel coordinates as well as the color you want to use. An example will (hopefully) make this clear.

 1 #! /usr/bin/env python
 2
 3 # Plot random pixels on the screen.
 4
 5 import pygame
 6 import random
 7
 8 # Window dimensions
 9 width = 640
10 height = 400
11
12 screen = pygame.display.set\_mode((width, height))
13 clock = pygame.time.Clock()
14 running = True
15
16 **while** running:
17     x = random.randint(0, width-1)
18     y = random.randint(0, height-1)
19     red = random.randint(0, 255)
20     green = random.randint(0, 255)
21     blue = random.randint(0, 255)
22
23     screen.set\_at((x, y), (red, green, blue))
24
25     **for** event **in** pygame.event.get():
26         **if** event.type == pygame.QUIT:
27             running = False
28
29     pygame.display.flip()
30     clock.tick(240)
31

This program just plots random pixels until the window is closed. It should be fairly straight-forward.

**Exercises**

1. Modify the program above to take the window dimensions as command line arguments.
2. Modify the program to start with a different background color.
3. Write a program that contains a list of 16 preset colors. You can choose any colors you want, but red, green, blue, yellow, brown and purple are a few good ones. Create a program that plots pixels and random coordinates and uses random colors out of the preset list of colors only.
4. Write a program that plots a pixel and the mouse position when the user presses the left mouse button.

**Keydown events**

You have learned how to deal with the QUIT event as well as MOUSEMOVE, MOUSEBUTTONDOWN and MOUSEBUTTONUP. This time let’s look at the KEYDOWN event. This even occurs when the user presses a key down.

When we recieve a KEYDOWN event, the event object will hold the code of the key that was pressed in an attribute called ‘key’. So by comparing event.key to whichever key we are interested in, we can find out if that was the actual key that was pressed. As usual, since I’m not good at explaining things, I hope the code below will make it clear.

 1 #! /usr/bin/env python
 2
 3 # Move a single pixel around the screen without crashing against the borders.
 4
 5 import pygame
 6
 7 # Window dimensions.
 8 width = 640
 9 height = 400
10
11 # Position of the pixel.
12 x = width / 2
13 y = height / 2
14
15 # Direction of the pixel.
16 dir\_x = 0
17 dir\_y = -1
18
19 screen = pygame.display.set\_mode((width, height))
20 clock = pygame.time.Clock()
21 running = True
22
23 **while** running:
24     x += dir\_x
25     y += dir\_y
26
27     **if** x <= 0 **or** x >= width **or** y <= 0 **or** y >= height:
28         **print** ("Crash!")
29         running = False
30
31     screen.fill((0, 0, 0))
32     screen.set\_at((x, y), (255, 255, 255))
33
34     **for** event **in** pygame.event.get():
35         **if** event.type == pygame.QUIT:
36             running = False
37         **elif** event.type == pygame.KEYDOWN:
38             **if** event.key == pygame.K\_UP:
39                 dir\_x = 0
40                 dir\_y = -1
41             **elif** event.key == pygame.K\_DOWN:
42                 dir\_x = 0
43                 dir\_y = 1
44             **elif** event.key == pygame.K\_LEFT:
45                 dir\_x = -1
46                 dir\_y = 0
47             **elif** event.key == pygame.K\_RIGHT:
48                 dir\_x = 1
49                 dir\_y = 0
50
51     pygame.display.flip()
52     clock.tick(120)

This program moves a pixel along the screen. Using the cursor keys you can change the direction of the pixel. If the pixel hits any of the borders the program terminates. Notice that keep a track not only of the x and y coordinates, but also the horizontal and vertical direction. When you need to keep track of several attributes of a single “thing” like this, that thing is definitely a candidate for being turned into a class.

In order to get used to moving “things” out into being classes, let’s take a look at an example of this. The following example looks and behaves just like the previous one, but there is a difference. We now create a class called MovingPixel. This class is responsible for keeping its own internal state (its current position, horizontal and vertical direction, etc). We could have added many more attributes to the class, but for the time being we keep things as simple as possible.

The class also contains a few methods in order for the main program to communicate to the class to perform certain actions, such as change direction, draw itself onto the specified surface and so on.

All in all, our program has grown slightly longer. But notice that the main program has grown a little less complex because we moved some things that are internal to the pixel into the MovingPixel class.

This example is obviously very simplistic and I don’t necessarily recommend that you create classes to represent pixels (although to be fair, ours is a *moving* pixel). But the principle can be used on more complex objects that maybe require many drawing operations to draw. We will see more on these in future tutorials.

 1 #! /usr/bin/env python
 2
 3 # Move a single pixel around the screen without crashing against the borders.
 4
 5 import pygame
 6
 7 # These are used for directions.
 8 UP = (0, -1)
 9 DOWN = (0, 1)
10 LEFT = (-1, 0)
11 RIGHT = (1, 0)
12
13 **class** MovingPixel:
14     """ A moving pixel class. """
15
16     **def** \_\_init\_\_(self, x, y):
17         """ Creates a moving pixel. """
18         self.x = x
19         self.y = y
20         self.hdir = 0
21         self.vdir = -1
22
23     **def** direction(self, dir):
24         """ Changes the pixels direction. """
25         self.hdir, self.vdir = dir
26
27     **def** move(self):
28         """ Moves the pixel. """
29         self.x += self.hdir
30         self.y += self.vdir
31
32     **def** draw(self, surface):
33         surface.set\_at((self.x, self.y), (255, 255, 255))
34
35 # Window dimensions.
36 width = 640
37 height = 400
38
39 screen = pygame.display.set\_mode((width, height))
40 clock = pygame.time.Clock()
41 running = True
42
43 # Create a moving pixel.
44 pix = MovingPixel(width/2, height/2)
45
46 **while** running:
47     pix.move()
48
49     **if** pix.x <= 0 **or** pix.x >= width **or** pix.y <= 0 **or** pix.y >= height:
50         **print** ("Crash!")
51         running = False
52
53     screen.fill((0, 0, 0))
54     pix.draw(screen)
55
56     **for** event **in** pygame.event.get():
57         **if** event.type == pygame.QUIT:
58             running = False
59         **elif** event.type == pygame.KEYDOWN:
60             **if** event.key == pygame.K\_UP:
61                 pix.direction(UP)
62             **elif** event.key == pygame.K\_DOWN:
63                 pix.direction(DOWN)
64             **elif** event.key == pygame.K\_LEFT:
65                 pix.direction(LEFT)
66             **elif** event.key == pygame.K\_RIGHT:
67                 pix.direction(RIGHT)
68
69     pygame.display.flip()
70     clock.tick(120)

**Exercises**

1. Modify the above program so that the pixel can move diagonally as well. You could use q (K\_q) for up/left, e (K\_e) for up/right, y (K\_y) for down/left and c (K\_c) for down/right.
2. See if you can figure out how to make the pixel leave a trail.
3. See if you can figure out how to turn the pixel into a worm. Choose a length for the worm and make sure the body always follows the head of the worm (as it does in all those worm games.

**Conclusion**

We have reached the end of another tutorial. This one is maybe a little shorter than some of the previous ones. I think it’s better to keep each tutorial short and instead try to write them more frequently. What do you think? Please let me know.

We can have a whole lot of fun with pixels. In the next tutorial we will continue with pixels a bit more but also combine them with a few other drawing methods. We will look at the simplest form of collision detection which allows you to determine if two objects are in contact with each other. That could represent a crash, or the fact that the player has been “caught” or whatever.