## Pygame tutorial #8: the worm game

After the little detour I took with tutorial #7, we are going back to writing our little worm game. I am going to write a first, (barely) playable version here, and then we are gradually going to improve upon it.

As always, I am taking things really slow. That way I hope everybody can keep up. There are loads of other, more advanced pygame tutorials floating around for those who feel I’m moving too slowly. Many of them are really good.

**Rules of the game**

The idea behind the game is very simple. You control a worm that move around the screen looking for food. When the worm gets the food, you get some points and the worms grows longer. Crashing onto yourself or the borders ends the game.

I first played the worm game on a VIC-20, which means this was one of the first games I ever played.

**The Worm class**

We will expand a little upon the worm class the we developed earlier. I’m going to take it method by method. You will find the whole program at the end of the tutorial.

Let’s look at the constructor first. Here we just set up some information.

 1 **def** \_\_init\_\_(self, surface):
 2     self.surface = surface
 3     self.x = surface.get\_width() / 2
 4     self.y = surface.get\_height() / 2
 5     self.length = 1
 6     self.grow\_to = 50
 7     self.vx = 0
 8     self.vy = -1
 9     self.body = []
10     self.crashed = False
11     self.color = 255, 255, 0

This constructor only takes the surface as an argument. We position to worm in the center of the screen initially (remember the x and y attributes hold the position of the worm’s head). The worm’s initial length is 1. I have added a new attribute, grow\_to. As long as length is less than grow\_to our worm hasn’t reached full length yet. This will allow the worm to grow, something that is important in the game.

Apart from that, most things should be straight-forward. The vx (horizontal velocity) and vy attributes control the direction of the worm. Each one of these can be -1, 0 or 1. Depending on the direction of motion. The color attribute should be obvious. The body attribute is an array holding all the positions of the worm’s body. Crashed is a flag that gets set to true if the worm bumps into itself.

The worm’s event() method is hopefully clear now. It looks like this:

 1 **def** event(self, event):
 2     """ Handle keyboard events. """
 3     **if** event.key == pygame.K\_UP:
 4         self.vx = 0
 5         self.vy = -1
 6     **elif** event.key == pygame.K\_DOWN:
 7         self.vx = 0
 8         self.vy = 1
 9     **elif** event.key == pygame.K\_LEFT:
10         self.vx = -1
11         self.vy = 0
12     **elif** event.key == pygame.K\_RIGHT:
13         self.vx = 1
14         self.vy = 0
15

The move() method has changed slightly. Let us look at what it does.

 1 **def** move(self):
 2     """ Move the worm. """
 3     self.x += self.vx
 4     self.y += self.vy
 5
 6     **if** (self.x, self.y) **in** self.body:
 7         self.crashed = True
 8
 9     self.body.insert(0, (self.x, self.y))
10
11     **if** (self.grow\_to > self.length):
12         self.length += 1
13
14     **if** len(self.body) > self.length:
15         self.body.pop()
16

First we move the worm’s head to its new position. Where this is will depend on the values of vx and vy (note that exactly one of these will be non-zero). Next we check if the new position of the worm’s head exists in the worm’s body array. If it does, we have crashed onto ourselves and set the crashed flag.

The new position then gets inserted into the worm’s body array. We then check if the worm is growing. If it is, simply increase the worm’s length by 1. Finally, if we have reached full length, then we need to remove the last item of the body array. Make sure you understand how the different parts of this method work together to make the worm move and grow. See if you can figure out a few ways to improve it (it shouldn’t be too difficult once you’ve got the point of the code).

The draw() method should be familiar by now. It just plots out each one of the points in the worm’s body array.

1 **def** draw(self):
2     **for** x, y **in** self.body:
3         self.surface.set\_at((x, y), self.color)
4
5

There is a new method that I’ve called position(). This will return the x and y-coordinate of the worm’s head. This is just there for convenience.

1 **def** position(self):
2     **return** self.x, self.y
3
4

Another new method is the eat() method. This just instructs the worm to start growing.

1 **def** eat(self):
2     self.grow\_to += 25
3

That is how the Worm class works. That main thing that is new about it as compared to the one we did in tutorial #5 is that this worm has the ability to grow. In order to grow, this worm needs food, so let’s look at that next.

**The Food class**

The Food class needs a surface to draw itself onto, just like the Worm class. Apart from that it holds it x and y coordinates and color. It contains a method to draw itself and one to get its position in the form x, y. It looks as follows.

 1 **class** Food:
 2     **def** \_\_init\_\_(self, surface):
 3         self.surface = surface
 4         self.x = random.randint(0, surface.get\_width())
 5         self.y = random.randint(0, surface.get\_height())
 6         self.color = 255, 255, 255
 7
 8     **def** draw(self):
 9         self.surface.set\_at((self.x, self.y), self.color)
10
11     **def** position(self):
12         **return** self.x, self.y
13

When a new Food is created, it places itself at a random location on the surface it gets passed.

**The game**

Putting our two classes to use, we come up with this:

  3 # A simple worm game.
  4
  5 import pygame
  6 import random
  7
  8 **class** Worm:
  9     **def** \_\_init\_\_(self, surface):
 10         self.surface = surface
 11         self.x = surface.get\_width() / 2
 12         self.y = surface.get\_height() / 2
 13         self.length = 1
 14         self.grow\_to = 50
 15         self.vx = 0
 16         self.vy = -1
 17         self.body = []
 18         self.crashed = False
 19         self.color = 255, 255, 0
 20
 21     **def** eat(self):
 22         self.grow\_to += 25
 23
 24     **def** event(self, event):
 25         """ Handle keyboard events. """
 26         **if** event.key == pygame.K\_UP:
 27             self.vx = 0
 28             self.vy = -1
 29         **elif** event.key == pygame.K\_DOWN:
 30             self.vx = 0
 31             self.vy = 1
 32         **elif** event.key == pygame.K\_LEFT:
 33             self.vx = -1
 34             self.vy = 0
 35         **elif** event.key == pygame.K\_RIGHT:
 36             self.vx = 1
 37             self.vy = 0
 38
 39     **def** move(self):
 40         """ Move the worm. """
 41         self.x += self.vx
 42         self.y += self.vy
 43
 44         **if** (self.x, self.y) **in** self.body:
 45             self.crashed = True
 46
 47         self.body.insert(0, (self.x, self.y))
 48
 49         **if** (self.grow\_to > self.length):
 50             self.length += 1
 51
 52         **if** len(self.body) > self.length:
 53             self.body.pop()
 54
 55     **def** draw(self):
 56         **for** x, y **in** self.body:
 57             self.surface.set\_at((int(x),int(y), self.color))
 58
 59     **def** position(self):
 60         **return** self.x, self.y
 61
 62 **class** Food:
 63     **def** \_\_init\_\_(self, surface):
 64         self.surface = surface
 65         self.x = random.randint(0, surface.get\_width())
 66         self.y = random.randint(0, surface.get\_height())
 67         self.color = 255, 255, 255
 68
 69     **def** draw(self):
 70         self.surface.set\_at((self.x, self.y), self.color)
 71
 72     **def** position(self):
 73         **return** self.x, self.y
 74
 75 w = 500
 76 h = 500
 77
 78 screen = pygame.display.set\_mode((w, h))
 79 clock = pygame.time.Clock()
 80
 81 score = 0
 82 worm = Worm(screen)
 83 food = Food(screen)
 84 running = True
 85
 86 **while** running:
 87     screen.fill((0, 0, 0))
 88     worm.move()
 89     worm.draw()
 90     food.draw()
 91
 92     **if** worm.crashed:
 93         running = False
 94     **elif** worm.x <= 0 **or** worm.x >= w - 1:
 95         running = False
 96     **elif** worm.y <= 0 **or** worm.y >= h - 1:
 97         running = False
 98     **elif** worm.position() == food.position():
 99         score += 1
100         worm.eat()
101         **print** ("Score: %d" % score)
102         food = Food(screen)
103
104     **for** event **in** pygame.event.get():
105         **if** event.type == pygame.QUIT:
106             running = False
107         **elif** event.type == pygame.KEYDOWN:
108             worm.event(event)
109
110     pygame.display.flip()
111     clock.tick(240)

We initialize a counter called score to 0 (unless you want to cheat of course). We create a our worm and an initial food item.

In the game loop we need to make sure we draw not only the worm but the food item as well. This is done with food.draw().

In addition to checking if the worm has crashed with itself or the borders, we check if the worm has eaten the food. This is where the position() methods in the Worm and Food classes comes in handy. If the worm’s head is at the same location as the food the worm has taken the food. We then increase the score and print the new score to the terminal (which sucks, I know, but don’t worry one of our improvements will be to get the score onto the game screen). We also need to signal the worm to start growing. We do that by callingworm.eat(). Finally, since the food item has been eaten, we need to generate a new one.

The rest of the program should be familiar by now.

**Running it**

The one thing you will notice quickly with this game is how annoyingly difficult it is to eat the food (unless you have really good eye-sight which I don’t). This will be the first improvement we have to make to the game.

**Exercises**

1. Ew! There are like magic numbers, EVERYWHERE! Find each one and change it. It’s often useful to create attributes. For instance the worm is hard-coded to grow 25 pixels longer each time it eats. You could change that to an attribute called grow\_by (or whatever you like). That means it can be manipulated from outside the class.

2. Play around with the initial worm size and the amount that the worm grows by when it eats. By modifying these you can affect how difficult the game is when it first starts and how much more difficult it gets over time.

3. Currently the program ends as soon as the worm crashes. Change it so that the user gets asked if they want to play again.

4. Build upon exercise 2 to keep a track of the high score.

5. Imagine you wanted to turn this into a two-player game. What would you need to change in order to make the Worm class re-usable?

6. Add the concept of lives to the game. The player should start off with three lives. Each time they crash they lose a life. When they run out, the game is over.

7. Let the worm start at a random position.

8. Suppose you wanted to implement difficulty levels. What things would you vary in order to make the game more or less difficult?

**Conclusion**

Now that we have a functional game, we need to make it properly playable as well. The first thing we need to do is to make the food item a little larger so that our worm stands a reasonable chance of getting it. It would be nice if the worm could make a “chomp” sound when it takes a bite as well. That will be the topic for the next tutorial.