pyGame Object-Oriented Lecture: Token Class

(Examples: Bouncing Ball Object Example, Rain Example)

UTILIZING OBJECTS IN MOVEMENT

I. Moving Entities

When we first learned about implementing movement on a pyGame surface, we simply stored several values in separate variables or a list which stored where an item was and its direction of movement between frames.

However, if you think about it, these separate variables all belong to a single object. Since our first, and perhaps most simply moving object was a circle, we can now take that logic, but encapsulate it in an object.

This entire lecture will be on the Token Class and its use in two separate programs: a rain simulation and a rewritten version of the bouncing ball program.

Our basic object, a token, needs to store the following information:

(x, y) coordinate of the center of the circle

(dx, dy) representing the movement between frames of the circle

r representing the radius of the circle

color representing the color of the circle

Based on this analysis, we can build our constructor for the Token class as follows:

```
def __init__(self,myx,myy,mydx,mydy,myr,mycolor):
    self.x = myx
    self.y = myy
    self.dx = mydx
    self.dy = mydy
    self.r = myr
    self.color = mycolor
```

II. Methods for the Token class

Perhaps the most obvious method is a move method, which changes the (x,y) coordinates of the object by (dx, dy). In addition, we want to be able to change the velocity and perhaps even set the velocity and color.

These methods are fairly straightforward and are included below:

```
# Call each frame.
def move(self):
    self.x += self.dx
    self.y += self.dy

# Adds addDX to dx, and adds addDY to dy
def changeVelocity(self,addDX,addDY):
    self.dx += addDX
    self.dy += addDY

# Directly set the velocity to newDX, newDY.
def setVelocity(self,newDX,newDY):
    self.dx = newDX
    self.dy = newDY

# Change the color of this token to newcolor.
def changeColor(self,newcolor):
    self.color = newcolor
```

Now, instead of having separate lines in main that change x and y, or even a regular method, now we have a instance method in the Token class which updates the object by one frame in the move method. The rest of the methods included are roughly setter methods, which set instance variables accordingly.

Recall in the bouncing ball example, we wanted to update the dx or dy based on whether or not a wall was hit. In some sense, we can say that this logic belongs to object, meaning that we should add instance methods to the Token class for them, one for each wall. Of course, two of these methods need to know either the width or height of the display surface. First, let's look at the bounceUp(self) and the bounceLeft(self) methods which need no extra information:

```
# Executes updating y for bouncing off the top wall.
def bounceUp(self):
    if self.y + self.dy < self.r:
        self.dy = -self.dy

# Executes updating dx for bouncing off the left wall.
def bounceLeft(self):
    if self.x + self.dx < self.r:
        self.dx = -self.dx
```

In our original code, this functionality was simply in our main function embedded in a larger if statement. But here, the appropriate logic is confined to an instance method and can be called appropriately on any token object.

Next, for bounceDown and bounceRight, we need to pass in the height and width, respectively so that the method knows the appropriate information to complete its task. Here is the code:

```
# Executes updating dy for bouncing off the bottom wall.
def bounceDown(self, SCREEN_H):
    if self.y + self.dy > SCREEN_H-self.r:
        self.dy = -self.dy

# Executes updating dx for bouncing off the right wall.
def bounceRight(self, SCREEN_W):
    if self.x + self.dx > SCREEN_W-self.r:
        self.dx = -self.dx
```

Since these methods might be called from various different places, it makes the most sense for it to take in the height and width of the screen, respectively. Again, this just encapsulates the logic previously discussed in the non-object oriented bouncing ball program.

III. Last Three More Complicated Methods

Now, to update a full frame for a program that has bouncing balls (Tokens), we can create our own updateFrame method, which calls both move() and each of the bounce methods we just looked at. For this to work, need to pass in the display surface to the method:

```
    # Update for a single frame.
    def updateFrame(self, DISPLAYSURF):
        self.move()
        self.bounceLeft()
        self.bounceRight(DISPLAYSURF.get_width())
        self.bounceUp()
        self.bounceDown(DISPLAYSURF.get_height())
        self.draw(DISPLAYSURF)
```

Notice that we just pass in the display surface so we can extract the width and height of it.

Our other method that will take in the display surface is the draw method, which will draw the object to the display surface. Here is that method, which just has a single pyGame method call.

```
    # Draws this object on the display surface as a circle.
    # Likely to be overridden most of the time.
    def draw(self, DISPLAYSURF):
        pygame.draw.circle(DISPLAYSURF,  self.color,  (self.x,
self.y), self.r, 0)
```

Finally, though we don't use this method in either of the two examples shown in this lecture, it absolutely makes sense to have a method which determines whether or not two Token objects have collided. This involves solving circle-circle intersection. Luckily of all intersections, this is perhaps the easiest to solve for mathematically.

We know that if two circles are barely touching at a single point, then the distance between the centers of the two circles equals the sum of the two radii. If that distance is less than the sum of the two radii, then the circles intersect with positive area. Otherwise, if the distances is greater than the sum of the two radii, there is no collision. Since we don't have to be perfect with collision since things happen so fast frame to frame (we just have to be correct within one frame), we can just

compare the distance squared between the two centers, and if this is less than or equal to the square of the sum of the two radii, then we have a collision. Notice that since the radius and x and y coordinates are all integers, that our code does no floating-point operations (meaning it's faster and more accurate):

```
# Returns true iff the two circles represented by self and
other intersect.

    # Should be overridden for different objects.
    def collide(self, other):
        center_dx = self.x - other.x
        center_dy = self.y - other.y
        dist_sq = center_dx*center_dx + center_dy*center_dy
        dist_sq_radii = (self.r + other.r)*(self.r + other.r)
        return dist_sq <= dist_sq_radii
```

First we take the difference in x and y between the two centers. Then if we square these differences, we get the distances squared between the centers. Then, we take the sum of radii and square that. Finally we return True if and only if the first computation is less than or equal to the second one.

IV. Bouncing Ball Object Version

You'll notice that much of the key logic from the original bouncing ball example is already present in the Token class. Of course, since that example had only one ball, to show how easy it is to reuse the code for multiple balls, this example allows the user to easily edit the code in a single spot so that there are as many balls as they want. Each ball is a Token object stored in a list. In the game loop, each ball is moved one frame and then displayed. Also, with more than one ball, we can test the collision code. One idea that isn't implemented is to make any two balls that collide disappear (or maybe just make the smaller one disappear). Another potential idea would be to have the balls bounce off of each other (this is a bit more complicated). Since we wanted to keep this example simple, all we'll do is print out to standard output (not the graphics screen) the fact that there is a collision and which two balls it's in between (their 0 based number in the list).

The code is organized as follows: The top of the file has imports and constants, followed by a single method which allows us to create a random integer in a range not equal to 0, followed by the main method which runs the program.

First let's just look at the imports:

```
import random
import math
import time
import pygame, sys

from TokenFile import token
from pygame.locals import *
```

The first few are typical. The only one that's new is the TokenFile import, which tells us that we're importing the token class from that file. This allows us to split up the program into two files. We'll formally look at this in more detail later, but this emphasizes the point that the token class is a stand-alone unit that can be reused in different settings.

Next we have some constants – these can of course be changed to suit the taste of the user:

```
# Useful Constants
SCREEN_W = 1000
SCREEN_H = 600
BALL_R = 15
NUM_BALLS = 10
```

Since we don't want any of the balls to be moving horizontally or vertically, we need to make sure we can easily generate a random number in a range that isn't an integer. Note that this isn't a task that belongs in any class, because this isn't necessarily a property of a token, which is why this isn't an instance method in the token class. Thus, it's just a regular function in the bounceballsplit.py file:

```
# Returns a random integer in between low and high != 0.
def myrand(low,high):
    res = 0
    while res == 0:
        res = random.randint(low, high)
    return res
```

It's up to the user to never call this function with both parameters set to 0, which would cause an infinite loop. As long as low to high is a reasonable range, the while loop isn't expected to run too often.

In the first part of main, we automatically generate NUM_BALLS number of balls and add each to the list. Each ball is randomly generated with a random position and a random direction of movement that is neither vertical nor horizontal. We use these numbers to call the token constructor and create a token object, which then gets added to the list, mytokens:

```
def main():

    # Basic Set Up
    pygame.init()
    DISPLAYSURF= pygame.display.set_mode((SCREEN_W, SCREEN_H))
    pygame.display.set_caption("Object Oriented Bouncing")

    clock = pygame.time.Clock()

    # Make NUM_BALLS random tokens.
    mytokens = []
    for i in range(NUM_BALLS):

        # Somewhere on the screen.
        x = random.randint(1, SCREEN_W-BALL_R)
        y = random.randint(1, SCREEN_H-BALL_R)

        # Random non-zero movement in both directions.
        dx = myrand(-2,2)
        dy = myrand(-2,2)

        mytok = token(x,y,dx,dy,BALL_R,pygame.Color("blue"))
        mytokens.append(mytok)
```

This code is fairly clean. We can make use of the token class easily by creating multiple token objects without added complexity.

Now, we get to the last part of the code, the game loop. Since there's no real game going on, all we'll do is process quitting (which we do in all of our pyGame programs) and then just draw our objects on the display surface, followed up updating the display. At the end, before moving onto the next frame, we'll look for collisions between all pairs of balls. Let's look at all of this code on the next page.

```
# Game loop.
    while True:

        # Look for events - we aren't using this right now.
        for event in pygame.event.get():

            if event.type == QUIT:
                pygame.quit()
                sys.exit()

        # White background.
        DISPLAYSURF.fill(pygame.Color("white"))

        # Essentially update each ball.
        for item in mytokens:
            item.updateFrame(DISPLAYSURF)

        # Update what we put on the canvas.
        pygame.display.update()

        # Just testing collision code.
        for i in range(len(mytokens)):
            for j in range(i+1, len(mytokens)):
                if mytokens[i].collide(mytokens[j]):
                    print("collide",i,j)

        # Wait a bit!
        clock.tick(100)

# Run it
main()
```

Since we already have the updateFrame method, we just have to call that for each token object, before we update the display. Then, we've included some test code that just calls our collision function to help us determine if it's working or not.

V. Rain Example

In the rain example we have a list of token objects again, but this time they are raindrops falling from the sky. Thus, our random direction of movement will always be down with the left, right, neither being selected randomly. Since we want to see lots of rain drops, the program generates in between 1 and 10 new drops, randomly to add to each frame to drop from the top. A set is used to make sure that no two drops start at the exact same place. (A set is similar to a list except that it doesn't allow for duplicates.)

Just like the previous example, we have several constants we're using:

```
# Useful Constants
SCREEN_W = 1000
SCREEN_H = 600
DY = 5
RADIUS = 10
```

Since we keep on adding rain drops, and eventually all the drops will fall below the screen, it's a good idea for us to remove the drops that will never show up on the screen again. To this end, we'll have a method that removes all of the useless raindrops. This method takes in the display surface height, so it knows which drops to remove. The other function simply moves all items in a list by calling the tokenFile's move method:

```
# This function handles moving each item listed in items.
def move(items):
    for item in items:
        item.move()

# This function removes all items that will never be visible
again.
def removeUseless(items, SCREEN_HEIGHT):
    for item in items:
        if item.y > SCREEN_HEIGHT:
            items.remove(item)
```

The main function is similar to the Bouncing Ball example. We start with an empty list. Inside the game loop, we'll add new raindrops to it each frame, followed by drawing all of the raindrops on the display surface and updating it.

We must do the following for processing:

1. Changing the velocity of each raindrop once every five frames (so the drops fall faster over time)
2. Move each drop.
3. Remove the useless ones.

```
def main():

    pygame.init()
    DISPLAYSURF = pygame.display.set_mode((SCREEN_W, SCREEN_H))
    pygame.display.set_caption("Let it rain!")
    clock = pygame.time.Clock()

    rain = []
    frame = 0

    while True:
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

        numNewRain = random.randint(1, 10)
        xvals = set()
        while len(xvals) < numNewRain:
            x = random.randint(1, SCREEN_W)
            xvals.add(x)

        for val in xvals:
            rain.append(token(x, 0, 0, DY, RADIUS,
                                        pygame.Color("blue")))

        DISPLAYSURF.fill(pygame.Color("white"))

        for item in rain:
            item.draw(DISPLAYSURF)
        pygame.display.update()

        if frame%5 == 0:
            for item in rain:
                item.changeVelocity(0,1)

        move(rain)
        removeUseless(rain, SCREEN_H)

        frame += 1
        clock.tick(30)
```

To create a set, we use the set constructor, setting xvals to a new set. If we add a duplicate to a set, it won't change the size of the set, which is a simple way and efficient way make sure each drop starts at a separate place. Here is that code segment:

```
        numNewRain = random.randint(1, 10)
        xvals = set()
        while len(xvals) < numNewRain:
            x = random.randint(1, SCREEN_W)
            xvals.add(x)
```

Once the set is finalized, we use it to generate and add all of the new raindrops to our main list:

```
for val in xvals:
    rain.append(token(x,0,0,DY,RADIUS,pygame.Color("blue")))
```

Finally, to make the rain look like regular rain (which speeds up some as it falls), we'll utilize the change velocity method to make each drop's velocity one higher than it was before, once every five frames:

```
        if frame%5 == 0:
            for item in rain:
                item.changeVelocity(0,1)
```

When you run the program, you can see the drops speeding down!