pyGame Lecture #2

(Examples: movingellipse, bouncingball, planets, bouncingballgravity)

MOVEMENT IN PYGAME

I. Realizing the screen is getting redrawn many times.

Let's take a look at the key portion of the previous program:

```
green = pygame.Color(0,255,0)

while True:

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

    DISPLAYSURF.fill(black)
    pygame.draw.ellipse(DISPLAYSURF, green, (500, 300, 50, 150), 10)
    pygame.display.update()
```

What's actually happening here is that the While True loop runs continuously, filling the display surface, drawing objects on the display surface and updating the display. In this particular example, the same exact ellipse and line are drawn over and over again, as opposed to being drawn just once.

Since we have access to variables, if we drew a line or ellipse based on the value of variables, then each different time the display is updated, the line would be drawn in a different place.

II. Using variables to make sure that each time an object is redrawn, it gets redrawn somewhere different, giving the appearance of motion.

Let's do a very simple edit here, where instead of the ellipse x and y coordinates being set at 500 and 300, we set them to expressions based on variables:

```
x = 20
```

```
y = 100

while True:

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

    DISPLAYSURF.fill(black)
    pygame.draw.ellipse(DISPLAYSURF, green, (x, y, 50, 150), 10)
    pygame.display.update()

    x += 2
    y += 1
```

If we carefully trace this code, the first time through the while loop the ellipse is centered at x = 20, y = 100. But, after the display updates for the first time, we ***change*** both x and y so that x = 22 and y =101. Thus, the next time the ellipse is drawn, it's drawn 2 pixels to the right and 1 pixel down compared to where it was drawn previously. Because we initially fill the surface black each time through the while loop, the previously drawn ellipse with center (20, 100) doesn't show after the second update. Instead, as the while loop runs many times, it appears as if the ellipse is traveling in a straight line, roughly from the top left to the bottom right of the screen.

In essense, the intial position of the ellipse is (20, 100) and in between each frame, the ellipse is translated 2 pixels in the x direction and 1 pixel in the y direction. In physics we call this, (2, 1), the velocity vector. The 2 represents the change in x and the 1 represents the change in y. In both physics and mathematics classes, one standard notation refers to the change in x as dx and the change in y as dy.

III. What happens if we don't wipe the display surface clean?

Now, let's take the previous answer and just move the line that paints the display surface all black to be ***before*** the while True loop:

```
x = 20
y = 100
DISPLAYSURF.fill(black)
while True:

    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
    pygame.draw.ellipse(DISPLAYSURF, green, (x, y, 50, 150), 10)
    pygame.display.update()

    x += 2
    y += 1
```

After a second or so, what we see is the following static image:

<Insert Image Here>

Essentially what has happened is that as the loop ran many times, each time we added one ellipse to the canvas, without ever painting over the whole thing. Over time, all of these overlapping ellipses looked like a wide diagonal line.

IV. Getting an object to "wrap around" the screen.

In the first example with movement, we showed a strategy to move an object in the same direction at the same speed (in physics this is called constant velocity) for the duration of our program. The strategy is to set two variables representing the x coordinate and y coordinate of our object to some initial value before our main game loop. Then, for each time the game loop runs, we add/subtract a constant amount from each, representing the change in both x and y we would like to see between frames.

Unfortunately, as was seen with the ellipse example, with this sort of movement, we'll lose an object forever once it goes off the screen.

One possible modification to the movement of an object to make things more interesting is to have the object "wrap around" the screen. If the object goes off the bottom of the screen, have it reappear at the top. Similarly, if an object goes off the right side of the screen, have it reappear at the left. Do similar wrap arounds in the opposite direction as well.

In some sense, the key is to detect when we've gone off the left, right, top or bottom of the screen. If we detect this behavior, then we have to translate the object (but not change its dx or dy) to the opposite side of the screen.

For example, given that our screen is 1000 pixels wide, the following edit takes care of the right to left transition:

```
if x > 1000:
     x = 0
```

Three more if statements handle the other three transitions. Alternatively, if we carefully understand mod, we can replace the statement above with a single statement with an if:

```
x = (x + dx)%1000
```

Essentially, when we mod by 1000, if something goes over 1000 by a small amount, it subtracts 1000 from it.

The main part of our new program is as follows:

```
x = 20
y = 100
dx = 2
dy = 1

while True:

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

    DISPLAYSURF.fill(black)
    pygame.draw.ellipse(DISPLAYSURF, green, (x, y, 50, 150), 10)
    pygame.display.update()

    x = (x + dx)%1000
    y = (y + dy)%600
```

V. Getting an object to bounce off a wall

To get an object to bounce off a wall, instead of translating it, we want to change the direction of its ***velocity***. Thus, we we reach the bottom wall, if we continue to have a positive dy value, the ball will go below the bottom of the screen. We can simply detect when this occurs and negate our dy value. Similarly, if we go off the screen to the top, we must flip our dy value from negative to positive. In both instances, the same line of code does the trick:

```
dx = -dx
```

Instead of using such a big ellipse, lets just use a regular circle that looks like round ball and make that ball bounce off the four walls of the screen. The resulting program is as follows:

```
# Our initial settings
x = 10
y = 10
r = 10
dx = 2
dy = 2
```

```
while True:

    # We just look to see if the user wants to exit.
    for event in pygame.event.get():

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

    # Translate our object for the next frame.
    x += dx
    y += dy

    # This does our bounce by changing the velocity component.
    if x >= 1000-r or x <= r:
        dx = -dx
    if y >= 600-r or y <= r:
        dy = -dy

    # Draw, update and wait!
    DISPLAYSURF.fill(black)
    pygame.draw.circle(DISPLAYSURF, red, (x, y), r, 0)
    pygame.display.update()
    clock.tick(80)
```

What happens here is that if we get to any boundary wall, all we have to do is flip that particular direction of movement (either dx or dy). The two if statements after the translation take care of this change of the movement direction. (Again, notice that x and y are NOT changed in this if statement since we don't want the ball to "jump" to a different place on the screen, but simply move in a different direction from where it is now.)

VI. Circular movement (Trigonometry Required)

One type of common motion is movement in a circle. Though this is a basic motion that we are familiar with from childhood on, the mathematics necessary to describe it isn't so elementary. The most typical way to describe circular movement is to use the sin and cos function, which themselves are defined based on the unit circle. In particular, when the angle θ is measured in radians, cos θ is defined as the x-coordinate of the point on the unit circle at angle θ, as measured counter-clockwise from the positive x-axis and sin θ is defined as the y-coordinate of the point on the unit circle at angle θ. For a pixel system, if our center is at (cx, cy) and the radius of the circle of movement is rm, then we can describe the x and y values that correspond to any angle theta as follows:

x = cx + (rm)cos(theta)

y = cy + (rm)sin(theta)

We can simply change theta after each frame, recalling that a change of 2π in theta represents a full revolution. With some trial and error, we can see what rate of change we would like.

In the following example we draw a sun in the middle with three orbiting planets. The planets' in this example will complete a revolution in proportion to the area of the circle they define (via revolution). This means that a planet that is twice as far from the sun as another planet will take 4 times as long to go all the way around the sun.

The program is included in its entirety. This example was made to be slightly more accurate than previous examples. As a result, the code is more lengthy. After the code, each piece will be explained.

```
import math
import pygame, sys
from pygame.locals import *

pygame.init()
DISPLAYSURF = pygame.display.set_mode((700, 700))
pygame.display.set_caption("Planets")

black = pygame.Color(0,0,0)
red = pygame.Color(255,0,0)
purple = pygame.Color(255,0,255)
orange = pygame.Color(255,165,0)
blue = pygame.Color(0,0,255)
clock = pygame.time.Clock()

# Revolution radii matching real ratios for Mercury, Venus, Earth
# which are .39, .72 and 1.
rRevMercury = 98
rRevVenus = 180
rRevEarth = 250

# These are relative radii of the planet themselves, but not
# the sun isn't relative. The real sun is much better, relatively.
rMercury = 10
rVenus = 23
rEarth = 25
rSun = 50

# All of our initial positions.
xSun = 350
ySun = 350

# Ratios of areas (just areas divided by pi...)
aMercury = rRevMercury*rRevMercury
aVenus = rRevVenus*rRevVenus
aEarth = rRevEarth*rRevEarth

# Current angles of each planet.
thetaMercury = 0
thetaVenus = 0
thetaEarth = 0

# Magic number that controls the speed of all the planets
# The planet speeds are relatively set.
dTheta = 150
```

```
while True:

    # We just look to see if the user wants to exit.
    for event in pygame.event.get():

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

    # Adjust the angle of each planet accordingly.
    thetaMercury += dTheta/aMercury
    thetaVenus += dTheta/aVenus
    thetaEarth += dTheta/aEarth

    #Calculate x and y of each planet accordingly.
    xMercury = (int)(xSun + rRevMercury*math.cos(thetaMercury))
    yMercury = (int)(ySun + rRevMercury*math.sin(thetaMercury))
    xVenus = (int)(xSun + rRevVenus*math.cos(thetaVenus))
    yVenus = (int)(ySun + rRevVenus*math.sin(thetaVenus))
    xEarth = (int)(xSun + rRevEarth*math.cos(thetaEarth))
    yEarth = (int)(ySun + rRevEarth*math.sin(thetaEarth))

    # Draw, update and wait!
    DISPLAYSURF.fill(black)
    pygame.draw.circle(DISPLAYSURF, orange, (xSun, ySun), rSun, 0)
    pygame.draw.circle(DISPLAYSURF, red, (xMercury, yMercury),
rMercury, 0)
    pygame.draw.circle(DISPLAYSURF, purple, (xVenus, yVenus), rVenus,
0)
    pygame.draw.circle(DISPLAYSURF, blue, (xEarth, yEarth), rEarth, 0)
    pygame.display.update()
    clock.tick(80)
```

Most of the first page is sets up constants. These constants are set up for a 700 x 700 screen and utilize some real information about the planets Mercury, Venus and Earth. The relative radii are accurate as well as the radii of revolution.What's not accurate about this simulation is that the actual paths of revolution trace ellipses and not circles. In addition, in order to more easily see the planets on a relatively small screen, the radius of the sun relative to the three planets is much smaller than it ought to be. Finally, instead of using the actual periods of rotation of the planets, the mathematical rule that approximates Kepler's Law related to planetary motion is used.

On the second page we first adjust the angles of rotation for each of the planets. These adjustments are different for each planet based on the total area encompassed by their revolutions. The value of dTheta was set via trial and error to make the revolution "look nice." Next, we do the most important calculations: the x and y

coordinates of each of the planets. The center of the sun represents the center of the circles traced by the paths of the planets, thus xSun and ySun represent this translation and are simply the added to the end result for each x and y coordinate respectively of the planets. Since cos and sin represent the x and y values of points on the unit circle, to scale these points to larger or smaller circles, we simply multiply cos and sin of the appropriate angle by the appropriate radius. In this case, for each planet, we multiply cos and sin of the appropriate angle by the radius of revolution for that planet. Once we've made all of these calculations, we have all the information we need to draw the appropriate circles for that frame.

VII. More Accurate Bouncing Ball (Physics Required)

When a ball is dropped from a height, the position of the ball is governed by the acceleration of gravity, wind resistance and the energy dissipated on the surface on which te ball bounces. While the latter two phenomena are relatively complicated to model, the effect of gravity on the ball is relatively easy to model. In this example we'll create a ball that bounces forever (perfect collisions with the ground with no energy lost), which follows the law of gravity.

In order for us accurately model the ball bouncing up, we have to keep track of the velocity of the ball. Let this be *v*. The relationship between velocity and acceleration is simply $v = v_0 + at$, where v_0 is the initial velocity. Acceleration for this example will simply be due to gravity. Since each iteration of the while loop is one time step and we are simulating the process, we will use the following approximation: $v_{t+1} = v_t + a$, calculating the new velocity in terms of the previous velocity. This will allow us to do a reasonable implementation without any variable t!

If we want to do a simple simulation however, it's easy enough for us to update our velocity after each simple time step and then assume that the ball travels at that velocity constantly just for that time step. When the ball hits the ground, we'll simply flip the sign of the velocity.

```
pygame.init()
DISPLAYSURF = pygame.display.set_mode((700, 700))
pygame.display.set_caption("Dropped Ball")

black = pygame.Color(0,0,0)
red = pygame.Color(255,0,0)
clock = pygame.time.Clock()

# Height will represent the traditional physics height
# The display is opposite with the y value further down being bigger.
# So, when we draw, we'll "flip" the height, so to speak.
height = 600

# This will be our initial velocity
velocity = 0

# This is constant - we have to play with this and the clock tick to
# get the simulation to look the way we want.
acceleration = -3

while True:

    # We just look to see if the user wants to exit.
    for event in pygame.event.get():

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

    # In one time step, this is what happens to velocity.
    velocity = velocity + acceleration

    # And this is what happens to height
    height = height + velocity

    # See if a bounce has occurred. If so, flip the velocity
    if height <= 0:
        velocity = -velocity-acceleration
        height = 0

    # Draw, update and wait!
    DISPLAYSURF.fill(black)
    pygame.draw.circle(DISPLAYSURF, red, (350, 700-height), 20, 0)
    pygame.display.update()
    clock.tick(50)
```

We adjust the velocity at each "time step" (loop iteration) based on acceleration. Then we adjust the height based on the velocity. Finally, if we've gone below the ground, we adjust our variables so the height is set back to 0 and the velocity flips signs. To keep the ball from losing "bounce" and converging to the ground, an extra update is given to the velocity in the if statement when the ball hits the ground, so that on its way up, the ball gets to its previous apex.