pyGame Lecture #3

(Examples: movement, tennispractice)

KEYBOARD INPUT IN PYGAME

I. Need for User Interaction

So far, all of our pyGame programs have not included any user interaction. Since the user doesn't interact with the computer, roughly speaking, each of these programs do the same exact thing each time they are run. We could infuse some random numbers into these programs so that each simulation is somewhat different, but ultimately, this wouldn't be too much fun either. Just like programs that run on the command line, the real fun occurs when the user interacts with the program and that interaction changes what the program does. Certainly, without this interaction, there would be no "game" in pyGame!

II. Detecting Keyboard Input in pyGame

Unlike traditional programming where we typically prompt the user for input and then wait until she types something in the keyboard, in a game, we'd prefer for other things to continue and simply to notice the presence of a key press and act accordingly. Namely, each time our code goes through its main game loop, we'd like to check if any key on the keyboard was pressed. A keypress is a type of event. In particular one of the events that we can check for in pyGame is the KEYDOWN event. If there is a KEYDOWN event, we can further check to see whick key was pressed. Each key has a code in pyGame. For example, the down arrow key code is K_DOWN. The variable that stores which key was pressed during a KEYDOWN event is key. A full list of the codes for each of the different keys can be found here:

https://www.pygame.org/docs/ref/key.html

Each of these codes can be checked for either a KEYDOWN event or a KEYUP event.

The following code segment checks to see if the up arrow key, down arrow key, left arrow key or right arrow key were pressed. If they were, the corresponding variable, x or y, is adjusted accordingly:

```
while True:

    for event in pygame.event.get():

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

        if event.type == KEYDOWN:

            if event.key == K_DOWN:
                y += dy
            if event.key == K_UP:
                y -= dy
            if event.key == K_LEFT:
                x -= dx
            if event.key == K_RIGHT:
                x += dx
```

Naturally, there are many events that we could check for in the event loop and a KEYDOWN event is the first non-trivial one that we've introduced. Many if statements could reside in the for event loop, each looking for various events.

III. Movement Example

The first example we'll use to illustrate checking for keydown events is a simple program that has a circle that the user can move with the arrow keys. The circle will start in the middle of the screen and sit there. Whenever the user presses the up key, the circle will move up, when the user presses the left arrow key, the circle will move left, and so forth. In this particular example, each single keydown press will result in a single movement of the circle. The circle won't continue moving if a key is held down and the circle x won't move again until the next keydown event. This gives the user very good control of the box, in terms of accuracy, but makes it very difficult to move the circle quickly. Codewise though, this example is relatively clean and simple without difficult mathematics. We utilize the concepts of storing an x and y coordinate for the circle's location as well as a dx and dy to represent its movement. But generally, dx and dy are 0 except for the game loop iterations where a keydown event occurs.

Here is the full program (movement):

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

pygame.init()
DISPLAYSURF = pygame.display.set_mode((600, 600))
pygame.display.set_caption("Movement Demo!")
black   = pygame.Color(0, 0, 0)
purple  = pygame.Color(255, 0, 255)

x  = 300 #The balls initial x position!
y  = 300 #The balls initial y position!
dx = 5   #How fast the ball is moving in the x
dy = 5   #How fast the ball is moving in the y

while True: #Game Loop

    for event in pygame.event.get():

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

        if event.type == KEYDOWN:

            if event.key == K_DOWN:
                y += dy
            if event.key == K_UP:
                y -= dy
            if event.key == K_LEFT:
                x -= dx
            if event.key == K_RIGHT:
                x += dx

    DISPLAYSURF.fill(black) #make the background black
    pygame.draw.circle(DISPLAYSURF, purple, (x, y), 20, 0)
    pygame.display.update() #Updates the frame
```

IV. Combining User Input with Interesting Movement

In some relatively simple video games, the user controls a paddle with two arrow keys (either left-right or up-down) and the goal is for the paddle to block a ball. In terms of separate components necessary to implement this general idea, we've already seen everything we need:

1. Linking key presses to changes in variables, which in turn display the paddle in different places.
2. Detecting when the ball reaches a paddle and then changing the direction of its velocity.

In this example we'll simulate tennis practice. In a regular tennis practice session, a coach hits balls over and over again to the student. The student runs to where the ball is and hits it back across the net. In this example the player will be on the left with the ability to slide their paddle all the way up or all the way down and the balls will be shot from the right side of the screen, traveling horizontally towards the left side of the screen. The goal will be for the player to simply line up their paddle so that the ball hits the paddle before getting past the left side of the screen. As the player hits more balls, the frequency with which the balls come will increase. For this program, when the paddle makes contact with the ball, the ball just disappears. (Many things are poorly done in this program. Improving it is left as an exercise for the reader!!!)

First, let's consider controlling the paddle. Rather than only have the paddle move when the arrow key is pressed down, we'll have the paddle continue moving. Namely, pressing the arrow key will affect the dy of the paddle and the paddle will continue in motion until the opposite key is pressed. Here is that segment of code:

```
while True:

    # Event loop for each iteration
    for event in pygame.event.get():

        # To handle exiting the game.
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        # Here are the key presses we respond to.
        if event.type == KEYDOWN:
            if event.key == K_DOWN:
                paddleDY += PADDLE_DELTA
            if event.key == K_UP:
                paddleDY -= PADDLE_DELTA
```

Unfortunately, when you press the same key a couple times, just changing dy makes the paddle very difficult to control. It goes very fast and it's hard to reign it back in. (Try it out without the second part of the code we're about to show!) Thus, in order to make the paddle a bit easier to control, we want to make the dy not immediately go to 0 (this is what happened in the movement example), but we'll have it slowly taper back to 0 in the absence of any key presses. Before our main game loop we set up a new variable called framecount:

```
framecount = 0
```

At the end of the game loop we simply increment framecount by 1:

```
framecount += 1
```

In the following code segment, we show how to reduce the absolute value of the paddleDY variable by 1 once every five frames. This code is located ***after*** the for loop looking for events:

```
if framecount%5 == 0:

    if paddleDY > 0:
        paddleDY -= 1
    elif paddleDY < 0:
        paddleDY += 1
```

Notice that if we want something to trigger once every k iterations of the main game loop, we can simply keep a framecounter and check its remainder when divided by k. This way, the taper doesn't happen too quickly (over five frames). Instead, after one key press, it takes 25 iterations of the game loop for the paddle to come to rest. (Depending on the clock tick, the total amount of time here varies, but this could easily be slightly less than one second.)

The other key pieces of logic to get this example to work are constraining the paddle movement so that it doesn't go off the screen, checking for a collision between the ball and paddle, and checking to see if the ball went off the left side of the screen. Let's investigate each of these pieces separately.

First, consider the problem of making sure the paddle doesn't move off the screen. If we know the y value near the top of the screen and the bottom of the screen, we can simply choose NOT to translate the rectangle corresponding to the paddle if it's already at the top or bottom:

```
if paddleY + paddleDY < SCREEN_H-PADDLE_L and paddleY + paddleDY >= 0:
    paddleY += paddleDY
```

Note that paddleY + paddleDY represents where we ***want*** to move the paddle. But, we will only move it there if this represents a valid location (not off the screen). Since the paddleY value represents the y value of the top of the rectangle, adding this to the length of the paddle (PADDLE_L) will give the y value of the bottom of the rectangle. What is checked above is equivalent. It states that the paddle only gets moved if the top of the rectangle will be at a y value less than the screen height minus the paddle length. Similarly, we require the y value of the top of the rectangle to be at least 0.

To check for a collision between the ball and paddle, this example makes a slight inaccuracy to simplify the arithmetic. Instead of treating the ball as a circle, it treats the ball as its circumscribed square, looking to see if that square intersects any part of the paddle. Checking for x is relatively simple becase the x value of the paddle is fixed.

To check for y, we must note that the center of the circle could be upto r pixels above the top of the paddle or upto r pixels below the bottom of the paddle. We need all three constraints (x value, and both the lower and upper bounds on y) to simultaneously be satisfied, so we use the and operator. If there is a collision, then we want to add 1 to the player's score and generate a new ball that will be hit to the player. The new ball will start on the right side of the screen at a randomly chosen y value:

```
if x < DELTA*3//2 and y + r >= paddleY and y - r <= paddleY+PADDLE_L:
    score += 1
    x = SCREEN_W - DELTA
    y = random.randint(DELTA, SCREEN_H-DELTA)
```

The first part of the if checks to see if the ball is close enough to the left end of the screen. The second part checks to see that the lower bound on the y coordinate of the ball is satisfied and the last part checks to see if the upper bound on the y coordinate of the ball is satisfied. y+r represents the bottom of the ball and y-r represents the top of the ball. We then compare these to the top and bottom of the rectangle, respectively.

Finally, checking to see if the ball went off the left side of the screen is easy; we just look for a negative x value:

```
if x < 0:
    print("Sorry, you lose, your score is",score)
    pygame.quit()
    sys.exit()
    break
```

There are many edits that could improve this "game." Here are a few ideas:

1. Make it a two player game, with a racket/paddle on the right side of the screen. Allow the ball to move diagonally somehow.

2. Don't end the game after one ball is missed. Allow the game to continue in some fashion, in either/both the one and two player versions.

3. Use actual tennis scoring for a single game with a serve triggered by a key press.

4. Come up with your own modification!