Pygame workshop part 1 Rendering, Game State, and Event handling

Initial setup:

Let's go over the *boilerplate code we will need to use pygame

*boilerplate is code that is always needed to setup a library, framework, or language in order to use to write new code that provides a solution you are seeking

First our imports tell python we will be using the libraries imported, and how they will be referred to in the program, because we are using default imports whenever we want to use something from the pygame library, or sys library we will need to refer to them by their name

```
#import pygame, and system library to use for rendering and quitting
import sys
import pygame
```

Here we tell pygame how to setup the window we will be rendering our game onto
Screen dim is a variable used to store the width, and height of the screen for our calculations later, and also for pygame to create a window of those dimensions

```
#init objects for pygame
screenDim = 300
```

The following two lines initialize all the data pygame needs so we can use it, including text rendering

```
pygame.init()
pygame.font.init()
```

The following line gets the pygame system clock so we can later set the frames per second we want pygame to render

```
FPS = pygame.time.Clock()
```

The following line creates a window of the dimensions provided for width, and height, and returns a surface which we can use to draw onto the window created.

```
#creates area to render on screen screenDimxscreenDim pixels
DisplaySurf = pygame.display.set_mode((screenDim,screenDim))

#defines frames per second
FPS.tick(25)
```

At this point your code should look like this:

```
#import pygame, and system library to use for rendering and 
quitting
import sys
import pygame

#init objects for pygame
screenDim = 300
pygame.init()
pygame.font.init()
FPS = pygame.time.Clock()

#creates area to render on screen screenDimxscreenDim pixels
DisplaySurf = pygame.display.set_mode((screenDim,screenDim))

#defines frames per second
FPS.tick(25)
```

Creating a game loop:

On some level every game you've ever played can be broken down into three parts:

Part 1 Taking input from the user:

This portion is fairly intuitive to understand on a high level. We want to allow the user to influence how the game changes over time, like in Mario when you press a particular button you can make Mario move left, or right, or jump up in the air.

In pygame we do this by handling events, to start we will need an event handler that will help us to wait the game when necessary

Every single iteration of the Game Loop we will use this for loop to iterate through all the events pygame has created. Pygame will create events for various types of input, in this tutorial we will review three of them, and the first is a quit event.

If the event type which we access with the identifier 'event.type' is the same as constant for pygame.QUIT then we know pygame has sent us an event requesting that we exit the application. To do this first we must call pygame.quit() this allows pygame to close the window, and give back any resources it has to the operating system to use for other processes, and then we can call sys.exit() to tell the python interpreter we wish to finish running our application.
Without both of these you will not quit properly, and may run into issues in the future

```
#basic game loop
while True:
#Defining event handlers
 for event in pygame.event.get():
#Handle quitting the game
  if (event.type == pygame.QUIT):
   pygame.quit()
   sys.exit()
```

Part 2 Updating some internal representation of the game:

The next portion is creating a representation of the game world that you can manipulate with code so that the game is dynamic, and not just an image on a screen.
Below we define the information relating to the player's position, size, and velocity, which is speed with a direction, in this case each velocity could be negative, or positive.
Because this is a part of the initialization of our game we call this like the other initialization functions for pygame outside the game loop

```
#define player information
playerX = 0
playerY = 0
playerXVel = 0
playerYVel = 0
playerDim = 50
```

Then each time we go through the loop we will be updating the playerX, and playerY values by adding the playerXVel to the playerX, and the playerYVel to the playerY, and later through event handling we can implement changing the player's velocity to make our player move around the screen
Let's add a section to our main game loop to handle this logic to update the internal representation of where the player is in the world:

```
#basic game loop
while True:
#Game logic here:
#move player position by vel each iteration of the game loop
 playerY += playerYVel
 playerX += playerXVel
```

For now this won't do anything because we need to implement user input to set the velocity using event handling, but we should make sure when we do we don't allow the player to leave the bounds of the screen, we can do this by ensuring the player's x is greater than 0, the player's x plus it's width is less than the width of the screen, in our case saved in the variable screenDim. This will make sure the player can't leave the screen, and go to the left, or right, but we still need to implement this to prevent the player from going up or down too far, and leaving the screen. Then if we detect the player is going up out of the screen

```
#detect collisions with walls to keep player on screen
 if(playerX < 0 or playerX + playerDim > screenDim or playerY <
0 or playerY + playerDim > screenDim):
  collision = True
```

```
#undo movement if collision detected
 if(collision):
  playerY -= playerYVel * deltaTime
  playerX -= playerXVel * deltaTime
```

Part 3 Creating visualizations of this internal representation:

Finally we need to take that virtual representation, and put it on the screen somehow, our game's output to the player. To do this to start we will use pygame to fill in the background each time with white so that we can immediately render the structures we have internally representing the game, in this case that is our player x, y, and width, and height
We call the function pygame.draw.rect in order to draw a filled rectangle, this is how we will blank out the screen each frame.
The function pygame.draw.rect takes 3 parameters, this looks like a lot, but it breaks down into relatively simple portions
First we have the display surface, or window we want to draw onto, if we had more than one window we would have to think about which window we want to render the game onto, but in this case we will simply render onto our only window which we reference by the variable DisplaySurf
The Second parameter is the color the rectangle should be, and pygame.Color is a function that takes a red, green, and blue value and returns a pygame color for the pygame.draw.rect function
The Third parameter is actually like a list, but in python we call it a tuple which consists of the x, and y positions, and width, and height in this case we will supply 0,0 for the x, and y, and the width, and height of the screen to the width and height to clear the entire screen each frame

```
#start of rerender here:
#Fill background with white each frame
#the last parameter is a tuple of 4 numbers(like a list)
representing the
# x, y, width, and height of the rectangle to be drawn in pixels
```

```
#pygame.Color allows us to create a color object that can store
red, green, and blue values between 0 and 255
 pygame.draw.rect(DisplaySurf, pygame.Color(255, 255, 255), (0,
0 , screenDim, screenDim))
```

Next we can render our representation of the player on the blank screen using the same function as follows:

```
#render player:
 pygame.draw.rect(DisplaySurf, pygame.Color(0, 0, 255),
(playerX, playerY, playerDim, playerDim))
```

Finally when we are done rendering our internal representation of the game to the screen we can tell pygame to render all those calls to pygame.draw.rect to the window by calling:

```
#Push updates in what has been rendered to the screen
 pygame.display.update()
```

Now that we have covered the basics…

Let’s move on to add some user input so we can move the player around the screen.

To do this we will use two new event types. pygame.KEYDOWN which indicates that a key on the keyboard has been pressed down, and pygame.KEYUP which indicates a key that was down has been lifted up.

```
pygame.KEYDOWN
pygame.KEYUP
```

And we will need to add some code to the event handling portion of the game using these events, and in addition to this we will need to be able to check what key was actually pressed down, and what key was lifted up, we will use the arrow keys to control the player’s motion, the so we need to compare the event.key variable from pygame to the pygame constants for the up, down, left, and right arrow keys, those event key constants are:

```
pygame.K_UP
pygame.K_DOWN
pygame.K_LEFT
pygame.K_RIGHT
```

When a key is pressed down we will update the velocity of the player accordingly, for instance if the right key is pressed we should set the x velocity of the player to 0.5
When left is pressed we should set it to -0.5
When the down key is pressed we should update the y velocity of the player to 0.5
When the up key is pressed we should update the y velocity of the player to -0.5
When any of these keys are released we should update the corresponding velocity to 0 to stop the player from moving

After implementing this your event handlers should look something like this:
#Defining event handlers

```
 for event in pygame.event.get():
#Handle quitting the game
  if (event.type == pygame.QUIT):
   pygame.quit()
   sys.exit()
  else:
   if (event.type == pygame.KEYDOWN):
#key pressed handlers
    if(event.key == pygame.K_DOWN):
     playerYVel += 0.5
    elif(event.key == pygame.K_UP):
     playerYVel -= 0.5
    if(event.key == pygame.K_RIGHT):
     playerXVel += 0.5
    elif(event.key == pygame.K_LEFT):
     playerXVel -= 0.5
#key released handlers
   elif event.type == pygame.KEYUP:
    if(event.key == pygame.K_DOWN or event.key == pygame.K_UP):
     playerYVel = 0
    if(event.key == pygame.K_RIGHT or event.key == 
pygame.K_LEFT):
     playerXVel = 0
```

And your project code should look like this:
#import pygame, and system library to use for rendering and quitting

```
import sys
import pygame

#init objects for pygame
screenDim = 300
pygame.init()
pygame.font.init()
FPS = pygame.time.Clock()

#creates area to render on screen screenDimxscreenDim pixels
DisplaySurf = pygame.display.set_mode((screenDim,screenDim))

#defines frames per second
FPS.tick(25)

#define player information
playerX = 0
playerY = 0
playerXVel = 0
playerYVel = 0
playerDim = 50

#basic game loop
while True:
#Game logic here:
#move player position by vel each iteration of the game loop
 playerY += playerYVel
 playerX += playerXVel
 
#pygame.Color allows us to create a color object that can store
red, green, and blue values between 0 and 255
 pygame.draw.rect(DisplaySurf, pygame.Color(255, 255, 255), (0,
0 , screenDim, screenDim))

#render player:
 pygame.draw.rect(DisplaySurf, pygame.Color(0, 0, 255),
(playerX, playerY, playerDim, playerDim))

#Push updates in what has been rendered to the screen
 pygame.display.update()

#Defining event handlers
 for event in pygame.event.get():
#Handle quitting the game
  if (event.type == pygame.QUIT):
   pygame.quit()
   sys.exit()
  else:
   if (event.type == pygame.KEYDOWN):
#key pressed handlers
    if(event.key == pygame.K_DOWN):
     playerYVel += 0.5
    elif(event.key == pygame.K_UP):
     playerYVel -= 0.5
    if(event.key == pygame.K_RIGHT):
     playerXVel += 0.5
    elif(event.key == pygame.K_LEFT):
     playerXVel -= 0.5
#key released handlers
   elif event.type == pygame.KEYUP:
    if(event.key == pygame.K_DOWN or event.key == pygame.K_UP):
     playerYVel = 0
    if(event.key == pygame.K_RIGHT or event.key ==
pygame.K_LEFT):
     playerXVel = 0
```

This is the end of part 1

By this point if you press run on the screen, click the pygame window, and then use the arrow keys you should be able to move around your player, and it should not be able to leave the window.