Pygame workshop part 2 obstacles, background elements, Python lists, and AABB collision detection

What we will be doing in this portion of the code is adding obstacles that the player cannot move through, as well as background elements rendered behind the player that the player can “walk over”.

To do this without letting the code get too confusing we need to learn about lists in Python. We will use lists to store the data that is relevant to the internal representation of each obstacle, and background element, this will be necessary for rendering them both, but also for the obstacles knowing where they are in the world is very important because that combined with the player’s information we can use that to detect if a player has hit one of these obstacles.

For everything we will render just like for the player we will need an x position, y position, width, and height, for convenience we will store each obstacle, and background element in an array of four elements, later we can add a fifth if we wish to add an image to render for each one, but for now, like the player they will be rectangles. In python we can declare an array like so:

```
obstacle = [50, 150, 50, 75]
```

This will create one obstacle record for us with as mentioned above an x, y, width, and height. The first element 50 is the x position in our representation, the second 150 is the y, the third 50 is the width, and the fourth 75 is the height.

Now we will put this in another list to make a two dimensional list, and add one more obstacle to demonstrate how we can handle rendering, and doing AABB (Axis Aligned Bounding Box which we will cover later) collision detection for multiple obstacles.

```
#define obstacles in game or collidable objects
obstacles = [[50, 150, 50, 75], [200, 100, 75, 50]]
```

The above defines a list of two obstacles in our game, and each one is a list with four elements for the x, y, width, and height. We have to update our rendering section next in order to get these on the screen, but first let’s also create a list of background elements in the same style right underneath this in the code:

```
#define background elements in game of non-collidable objects
backgroundElements = [[100, 50, 60, 75], [25, 140, 60, 25]]
```

Updating the rendering section of our code to draw these new obstacles, and background elements to the screen

So now we must update the code that renders our internal representation of the game because we have added new elements we need to render in each frame. This will be very similar to rendering the player, but because we have a list of obstacles, and a list of background elements we will need to use a for loop to go through each element of these lists calling the pygame.draw.rect method, and we will draw background elements in green, and obstacles in red.

So to begin rendering the obstacles we will write a for loop like this to go through each obstacle, and extract the x, y , width, and height from the inner list

#render player, and obstacles here:

```
 for obstacle in obstacles:
     obstacleX = obstacle[0]
     obstacleY = obstacle[1]
     obstacleWidth = obstacle[2]
     obstacleHeight = obstacle[3]
```

Once we have the data we need actually rendering an obstacle is just like rendering a player, we just need to do it inside the above for loop:

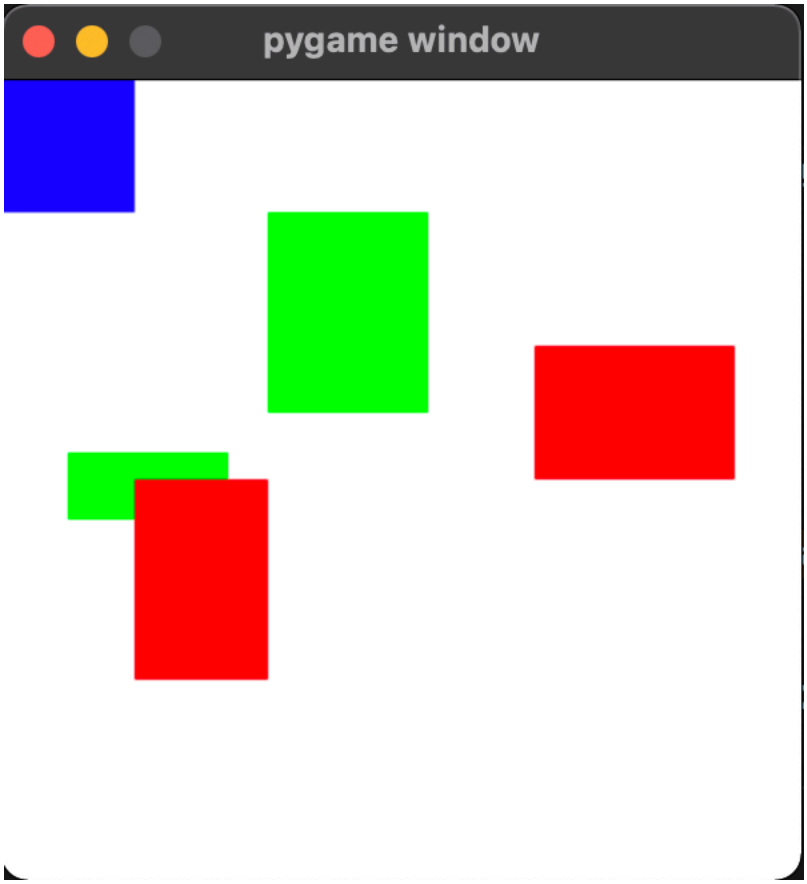
```
     pygame.draw.rect(DisplaySurf, pygame.Color(255,0,0),
(obstacleX, obstacleY, obstacleWidth, obstacleHeight))
```

We now need to render the background elements:

```
#render background here:
 for backgroundElement in backgroundElements:
     backgroundElementX = backgroundElement[0]
     backgroundElementY = backgroundElement[1]
     backgroundElementWidth = backgroundElement[2]
     backgroundElementHeight = backgroundElement[3]
     pygame.draw.rect(DisplaySurf, pygame.Color(0,255,0),
(backgroundElementX, backgroundElementY, backgroundElementWidth,
backgroundElementHeight))
```

Now when you run the game it should startup and run like this:

Where the Blue rectangle is the player, the green rectangles are the background elements, and the red are the obstacles, but wait! If you notice you can move right through the obstacles we have created, and that's no good. We must add collision detection next to fix this!



Updating the Game logic to add collision detection

Finally we will learn a bit about AABB collision detection, and implement it for our obstacles in the game logic section of our game loop so that when the player moves into one of the obstacles it's change in position due to it's velocity will be reversed this will simulate Newton's third law, for every action there is an equal and opposite reaction, and because this obstacle is currently an immovable object all the force the player exerts into moving into the obstacle will be pushed back at it causing it not to move this frame.

Let's talk about Axis Aligned Bounding Box or AABB collision detection in two dimensions.
If we assume that the two rectangles are not rotated at all then we can break collision detection up along two axises the x, and the y.

Let's first examine the x alone if I have two rectangles the first with an x of 0, and a width of 50, and the second with an x of 100, and a width of 50 also only analyzing to see if there is a possible collision on the x we can clearly see 0 + 50 is less than 100 or the highest x point of rectangle one is 50 which is less than the lowest x point of rectangle 2 so clearly there cannot be a collision, likewise if we moved rectangle 1 to have a starting x of 200 then it's lowest x is greater than the highest x of rectangle 2 which is 100 it's x position plus 50 it's width which is 150.

From this we can conclude that only when the highest x point in rectangle 1 is greater than the lowest x point in rectangle 2 and when the lowest point of rectangle 1 is less than the highest x point of rectangle 2.

Now this explanation was really for 1 dimension to extend it to two we ensure the same rules apply to the y axis

Now we already have a variable that we use to detect if a collision with the borders of the screen has occurred, and we already update the player position according to the above mentioned rules of physics, so the next thing is to also update that variable and set it to true if we detect a collision with an obstacle to prevent running through it, or even getting stuck inside it.

```
#detect collision with one obstacle that is rectangular
     if(playerX < obstacleX + obstacleWidth and playerX +
playerDim > obstacleX):
      if (playerY < obstacleY + obstacleHeight and playerY +
playerDim > obstacleY):
       collision = True
```

This is a good start.

But remember we have a list of obstacles so we need to go through each one extracting the relevant data using a for loop like when we rendered the obstacles, and background elements.

```
#check for collisions with every obstacle
 for obstacle in obstacles:

#get this obstacle's data
     obstacleX = obstacle[0]
     obstacleY = obstacle[1]
     obstacleWidth = obstacle[2]
     obstacleHeight = obstacle[3]

#detect collision with one obstacle that is rectangular
     if(playerX < obstacleX + obstacleWidth and playerX +
playerDim > obstacleX):
      if (playerY < obstacleY + obstacleHeight and playerY +
playerDim > obstacleY):
       collision = True
```

Now our AABB collision detection has been fully implemented!
Try to run the game and test it out.
You can also try to add or remove obstacles and background elements, or update their positions and dimensions, and see how it affects what you see in the “game”

Optional adding, and creating sprites for your game

*If you should choose to add rendering images/sprites for the you will be rendering those images instead of the rectangles, and make sure you use public domain, or open licensed graphics, or create your own, which you can do here:

https://sprite-creator.com/web/
Or
https://www.piskelapp.com/p/create/sprite

To add images to the background elements, obstacles or player we must add more data to our internal representation of the game, each obstacle must know what it's image is if you choose to add an image for one, and the same is true for the background elements.

In Pygame we can render png images instead of rectangles, let's update the player to use a png instead of a rectangle when we render, to start upload a png file to your repl

We will add this line of code to the player initialization:

```
playerImg = pygame.image.load('player.png')
```

You should replace player.png with whatever the name of your png file is.

We also want to scale this image to fit in the area of our player rectangle we can do this with the pygame.transform.scale function like so:

```
# Scale the image to your needed size
playerImg = pygame.transform.scale(playerImg, (playerDim, playerDim))
```

Then we only need to update how we render our player to finish, instead of using pygame.draw.rect we will use DisplaySurf.blit:

In the rendering section we will change this:

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

To this:

```
#render player:
 DisplaySurf.blit(playerImg, (playerX, playerY))
```

And that's it! You can do the same for the obstacles, and background elements, and continue to add more game logic to build your game from here!