Matthew Warren

HW1 Writeup

The first part of this assignment was just learning how to create a window with sfml. Most of the code needed was given in the tutorial webpage, so there were not many design decisions to make here. I just used the suggested size of 800x600 and the sf::Style::Default parameter to allow for resizing and closing. It is throwing a segmentation fault for me when closing which scared me for a second, but found out that is not something I have to worry about.

The design I had for drawing objects went through the most revisions when working on this project. I started with a simple structure:

- A platform class that inherited from the sf::RectangleShape class
- A moving platform class that inherited from the platform class
- A player class inheriting from sf::CircleShape.

In the platform and player classes, I added parameters in the constructor for their starting position and a path to the texture used for the object. This greatly simplifies creating these platform and player objects in the main loop, especially with all the error handling that goes into loading textures. For the moving platform class, I wanted to try and keep the code moving the platform in the class, not have it in the main function to reduce clutter there. I tried to keep it basic as I was still learning the ins and outs of c++/sfml, so I only gave it 2 directions for now, horizontal or vertical. The constructor took in an enum specifying one of the two, and a distance it would travel from its starting point. The platform will constantly move back and forth between the starting point and the distance specified. This felt like a good starting point for platforms that left room for growth. I was thinking by using an enum for direction that it would allow for more directions/styles in the future.

The next step was to add inputs. This would only affect the main function and player class. As with the moving platform, I wanted to keep as much out of the main loop as possible. In the player class, I added a function to assist with movement that takes in a keyboard input. So all the main loop needed was to check for the 4 directional inputs (WASD) and call the player move method with the appropriate key. The function in the player class determines what direction to move the player and does the action. There is not much special here now, but having a dedicated method for this will be useful when we get into more specialized movements such as jumping.

Collision was the most time consuming part of the assignment for me. My first attempt was to handle it in the player movement class, by passing in the other objects I wanted to check collision with and using the GetGlobalBounds() and intersect methods as described in the writeup. If a collision was detected, I tried to not allow movement in that direction anymore until they moved in the opposite direction. However, I found the problem with this was when it collided, all movement was stopped instead. The player just got stuck against the platform and I could not come up with a good way to fix it. After researching more into how collision usually works, I realized that if a collision is detected, I need to move the player back away from the object to resolve the collision. This was starting to turn into a lot of code to put into the single player movement method, and looking into the future I realized that there may be other things that I want to collide with other than a player and platforms. So I decided to restructure how I had my inheritance for shape objects. At the very top of my tree, I created a Collidable Object class that inherits from the sf::Rectangle shape, with player and platform now inheriting from this class. The only implications this had for those classes was that now the player was a rectangle, not a circle which ended up making detecting collisions easier. And now I will be able to create as many rectangle objects as I want with collision built in. In the collision class, I moved a lot of the position and texture functionality here since both player and platform were using it. I wrote a method to detect and resolve collisions, comparing the class it was called in and taking in another collidable object. After first detecting a collision using the same bounds and intersects methods from before, I then adapted an algorithm I found online to determine what side the collision was on. Once that was figured out, all I had to do was move the player right outside of the platform on the correct side. I think this is a great starting point for the project. The player can ride up a vertical moving platform which already feels a lot like a standard 2D platformer. Once more physics gets added in I'm sure it will need to be more fleshed out.