Jayden Sansom, jksanso2

Homework 1: SFML Foundations

Introduction

This assignment allowed for an introduction to SFML and set the base window and object structure for the continuing project. There were many decisions involved with implementation choices due to planning for future functionality. The three categories below represent parts two through four of the assignment, the first being excluded as it only was used to set up the base window, for which there were no design decisions to debate. The following sections will cover the decisions made and their results as well as how they were implemented with future functionality in mind.

Drawing Objects

Drawing objects in SFML is already a built-in feature, allowing me to add a window clear, draw objects, and display the new frame with the drawn objects in each loop while the window is open. While this is standard, the way that I chose to implement the object classes was more complex.

For the static platform, I had it extend the RectangleShape class and had its full constructor get a desired position, size, and texture path. The user can control these aspects of the object and in my demonstration, I chose to add it as a floor within the window.

The moving platform shared a similar constructor to its static counterpart, however, it also could get a destination position, a speed, and a pause length. This could allow the user to set a desired end location, how fast it will take to get there, and how long it will pause before reversing back to the original position. The movement all took place in the update function, using a direction vector to get how much the platform should move in consideration of time and speed towards the destination. This method resulted in a consistent direction movement and provided the user with more customization in how they would like their moving platform to operate.

The most important of the objects implemented was the Player class. It extends from the Sprite class to allow for more future capabilities when it comes to animation and other functions that are restricted to use by a Sprite object. At this time the player does not move as that will be added with the addition of user input.

Handling Inputs

Due to the restriction placed on the class, events were not an option while handling input from the user. Therefore, once designing this system, I chose to utilize the isKeyPressed function within the player update function. This allowed for movement to take place while a user is pressing the button rather than after the fact. I utilized this to create three input types, ‘A’ or Left to move the player left, ‘D’ or Right to move the player right, and ‘W’, Space, or Up to make the player jump.

Moving vertically was simple as it only required linear movement in the x or -x direction. I created a totalMovement private variable to hold a vector of floats that would be how much the player moves by. When the user aims to go left or right, the x of totalMovement is added or subtracted by the time times the speed of the player, reset every update frame back to zero.

Jumping was what caused some trouble as it was tricky to plan for gravity weighing the player back down. First, I had to make it so that the player can only jump once, then I set a new variable called jumpVelocity to be equal to the inverse of the player's jumpSpeed. Incidentally, this is because the x and y plane of the window work inversely to a common graph, therefore to move up, you must subtract rather than add when jumping, something that became a common error across my code. If the player is currently jumping, gravity will affect them by making the y of total movement be the set jumpVelocity times time then adding to jumpVelocity by gravity times the square root of time for effect in future loops. If the player has reached the floor, they will no longer be jumping and gravity will no longer affect the player. Finally, once all totalMovement has been calculated, the player will be moved by that amount.

Handling Collisions

Handling a collision was tricky to think about, especially when trying to emphasize efficiency and quality of appearance. My design initially began with creating a global list of all objects that have collision enabled, however, as I quickly learned, pointers to Sprites and Shapes could not be a part of the same vector list, even if abstracted. Therefore, I made an interface that would be inherited by my object classes, where it would act as the collider box and handle all collisions. This worked with much tweaking and I was able to add it within my Player update function to check for collisions, stopping movement if detected on that frame.

Once this was completed I noticed that if my player were to jump on the moving platform, once it would move, the player would remain static and unable to move while the platform moved within its bounding box. Additionally, once I jumped and hit the floor, it would not let me move vertically. This was solved by adding a resolveCollision function which would find the overlap between the colliding objects and set it to a position where it would no longer collide in the direction of the initial collision. This allowed my player to move with the platform while also allowing for movement on the floor.

Finally, I had many troubles with getting a player to move along with a moving platform. By adding references to the movement of what the player was colliding with I was able to get a semblance of what it should look like. However, when moving downward with a platform, it would no longer see collision as the platform updated to be outside of the player's bounds. This was solved by checking if it was previously on a platform, still above it, and isn't jumping. This allowed me to do an additional move function that solved this problem and got the movement to be smooth along with the platform.