Hyunsirk Choi

CSC 481

Dr. David Roberts

07 Sept 2022

Homework 1 Writeup

Section 0: Setup Grading Environment

I have been using WSL2 throughout the school year for multiple school coding works so I did not have to install the WSL2 again. I have successfully installed both SFML library and X11 and checked if the SFML display window was opening on a simple tutorial code.

Section 1: First SFML Project

I was able to create my first SFML project by referencing the tutorial codes from the SFML website. For this section, I was able to play around with the `sf::Event event` which I was able to change the title of the display window to change its title to the current window size. I was able to detect window resizing by checking `event.type == sf::Event::Resized`.

Section 2: Drawing Objects

I have created three rectangle-shaped objects by creating my own `CharacterShape`, `PlatformShape`, and `MovingPlatformShape` classes. All three objects are inheriting the `sf::Shape` class which I had to define `getPointCount()` and `getPoint()` functions. While I have been told that I was able to use more specific `sf::RectangleShape` class, it was interesting to learn how the `getPointCount()` and `getPoint()` functions work. The `getPointCount()` returns the number of points of the custom shape and I had to define where all of those points are positioned in `getPoint()`.

Another challenge in this section was to allow the `MovingPlatformShape` object to move around in a pattern. For simplicity, I have made the object move only left and right which will bounce off the wall. To achieve this, I had 4 conditions to check for the object to move in my intended pattern:

- check if moving left && check if the object reached the left wall;

  then change direction to right;
- check if moving left;

  then update the object position to move left a certain distance;
- check if moving right && check if the object reached the right wall;

  then change direction to left;
- check if moving right;

  then update the object position to move right a certain distance;

  All of these 4 conditions are in an if/else if statement in order which means only

one condition can be true at a time. I have also used a Boolean flag to represent the 2-way directions left and right.

Section 3: Handling Inputs

I have a total of 3 possible keyboard inputs to move the `CharacterShape` object and the inputs are Left Arrow, Right Arrow, and Spacebar for moving left, right, and jumping. I have created a Boolean flag for each input which the event handler will flip the flags when the key is pressed, and flip back to false if the keys are not pressed. With these Boolean flags, the separate if statements will update the object position accordingly. While it was simple enough to control the object to stay on the screen by ignoring the input on the wall, the jumping mechanic was much more complex.

The jumping mechanic is complex due to its overtime position change and making the motion smooth. To achieve the overtime position change with smooth motion, I have used a variable to keep track of the jumping time and used `M_PI` constant and `cos()` function from the `<cmath>` library. The jumping time variable `jumpTime` will be set to 0 when there is a jump input (Spacebar) and will be incremented 60 times meaning the jump motion will take 60 frames to execute its animation. Since the `jumpTime` variable is in frame counts, I have converted it into radius using `M_PI`. Then, I was able to feed the value to the `cos()` function. The `cos(M_PI * jumpTime / 60)` is then multiplied by a constant negative value to represent the moving distance and direction of the character while jumping. The code looks like this:

```
character.move(0, -JUMP_HEIGHT * std::cos(M_PI * jumpTime / JUMP_FRAME));
```

Section 4: Collision Detection

I have two platforms at this point which are `PlatformShape`, and `MovingPlatformShape`. Section 3 allowed me to jump and land on `PlatformShape` while any contact with `MovingPlatformShape` was ignored. I have realized it will be hard to use the same logic as Section 3 implementation since the implementation only works when the jump start and the jump end occur on the same y-axis. The problem is that the `MovingPlatformShape` is above the jumping platform and should land on a different y-axis. After learning about `getGlobalBounds()` and its `intersects()` functions, I have changed my implementation to stop the jump action as soon as the character and one of the platforms intersect; however, since the y-axis movement increments in a float value, the character will overshoot the y-axis value and end up inside the platform rather than having a clean land. Since it was inevitable to use float incrementation for the radius, I had to manually set the position of the character to land right on top of the platform as soon as it intersects. During the manual position setting, I had to position the character to be overlapped with the platform by a few pixels to make the `intersect()` function return true and represent that the character is “on” the platform.

Another challenge during this section was to implement the character movement that moves with the moving platform while the character is on top of the moving platform. Eventually, I decided to use a Boolean flag to indicate if the character is on the moving platform or not. If the character is on the moving platform, the character will receive the same exact movement as the moving platform which looks like the character is getting dragged around due to the friction on the moving platform

while it is actually the character moving at the same speed and direction as the platform. This provided the smoothest result and the character can still move on the moving platform.

Section 5: Scaling

The blue circle on the top right of the initial display screen represents the “Proportional Mode” where the rotating shape will change its size as the screen is resized. A click on the blue circle (the colored circle is actually a button) or any keyboard input will switch the circle color to red which represents the “Constant Mode” where the size of the rotating shape will now stay unchanged during a screen resize.

While this section sounded rather simple to implement, I had to do some research to avoid the shapes from squeezing/stretching and implement the button to work on resized screens. The squeezing/stretching issue was resolved using the `setView()` function inside my `sf::RenderWindow window`. On the other hand, while the screen did not look squeezed/stretched, the coordinate of the button before resizing the screen did not match the pixel location of the resized window so the button position was not working as I expected while implementing a button click. While keyboard input was enough and mouse button click was unnecessary, it was simple enough to use the `mapCoordsToPixel()` function from `sf::RenderWindow window` which allowed me to convert the old button coordinates into the current screen’s pixel locations.

```
// Mouse
sf::IntRect mouse(sf::Vector2i(event.mouseButton.x, event.mouseButton.y),
sf::Vector2i(1, 1));

// Button
sf::IntRect
relativeButton(window.mapCoordsToPixel(sf::Vector2f(toggleButton.getPosition()
.x, toggleButton.getPosition().y)),
window.mapCoordsToPixel(sf::Vector2f(TOGGLE_BUTTON_RADIUS * 2,
TOGGLE_BUTTON_RADIUS * 2)));
```

Conclusion

While I have learned a lot about how to use the SFML library for a simple platform game, I have realized some limitations of my implementation. The manual character positioning while landing on a platform seems to be a bad practice and should seek for a better implementation. Finding a new way to implement the smooth jumping motion may resolve this problem by avoiding float incrementations. Another limitation is that the platforms are not scalable since the behaviors of the two platforms are hard coded. As a game engine, it would need a solution to dynamically add new platforms without writing repeated codes on main.