Section 1: I have written my code to open the window to follow the specifications required in the project. In order to prevent screen tearing, I have decided to use the following function on my RenderWindow class: .setVerticalSyncEnabled(true). There is another method provided by the SFML documentation, although it is not recommended to use that one since it has been noted to be unreliable at times with the SFML implementation. The window continuously refreshes with a black screen within the main game loop, only ending once the close button has been pressed. The window follows default settings that allow resizing and closing.

Section 2-3: I created two additional classes to handle the platforms and character models for the game. The platforms class inherits from the RectangleShape class in SFML, while the player class inherits from the CircleShape class. Both the Platform and Player class are similar with the key differences being the specifics of the functions within. I decided to provide the Platform class the ability to move at angles. This was done by first calculating the unit vector of the platform when the orientation is first set (or recalculated whenever the orientation changes – this makes it so that the program doesn't have to redundantly recalculate the direction to travel). The calculation of these unit vectors required the usage of the math.h library (for sine to perform SOH CAH TOA). Next, the unit vector is multiplied by the speed property provided to the Platform. And finally, when the updatePosition() method is called by the main game loop, the Platform will be moved in the appropriate direction. The Platform also utilizes a cyclesWaited variable to keep track of the number of loops the main game loop has gone through, thus allowing a consistent animation. Although the Player class has a

very similar updatePosition method, it is still a shell in terms of animations. I was hoping in the future to be able to provide a cycling animation for the player model. Both the Player class and the Platform class have multiple constructors that allow easier construction of these objects. Although there are default speeds for the platforms and players to travel, I can easily change that by using one of the other constructors that specify a different speed or location.

Section 4: I handled collisions by dividing the handling between the Platform and Player classes. This was done by having the player “stop” moving in a direction that is occupied by another object (implemented by checking ahead of the area they’re trying to move into – preventing overlap in shapes), while having any platforms “push” any Player classes aside (I could make it so that the Platform can push other things aside, I’ve just had it do Players only for now). I was originally planning to try and implement a way to have the objects have their hitboxes land on top of each other rather than over, but eventually I deemed it too time-consuming and prioritized a way to prevent overlapping rather than frame perfect stops. The way I implemented this allows multiple Player classes to be pushed rather than any single one, thus allowing either multiplayer, or control over multiple entities. One of the caveats of the way I implemented the collision system though is that any object that is orientation/rotated at an angle that is not 0, 90, 180, 270, etc. will have janky hitboxes – if the object is rotated, it may appear as though I have collided with nothing.

Here I have snippets of code that I think are more difficult to easily locate and determine the purpose, thus I have isolated them here to specify their purpose.

Section 1-3:

Snippet of code from Platform that calculated movements that allow rotational movement of Platform.

```
this->setRotation(newOrientation);
this->pOrientation = newOrientation;
double yTemp = sin(getOrientation() * M_PI / 180) * (double) pLength;
double xTemp = sqrt(pow((double) pLength, 2) - pow(yTemp, 2));

yTravel = pSpeed * (float) (yTemp / (sqrt(pow(xTemp, 2) + pow(yTemp, 2))));
xTravel = pSpeed * (float) (xTemp / (sqrt(pow(xTemp, 2) + pow(yTemp, 2))));
```

Section 4:

Snippet of code from Platform that “pushes” multiple objects away.

```
if (cyclesWaited > getCycleWait()) {
	xTravel *= -1;
	yTravel *= -1;
	cyclesWaited = 0;
}
this->move(xTravel, yTravel);
cyclesWaited += (int)pSpeed;

bool *players = new bool[entities];
for (int i = 0; i < entities; i++) {
	players[i] = false;	// Initialize all elements to zero.
}

if (collisionTest(entityList, entities, players)) {
	for (int i = 0; i < entities; i++) {
		if (players[i])
			entityList[i]->move(xTravel, yTravel);
	}
}
delete[] players;
players = NULL;
```

Snippet of code from Player that “stops” overlap collision with another object.

```
// check ahead in direction
boundingBox.left += -pSpeed;
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left)
	&& !(collisionTest(entityList, entities, boundingBox, otherBox)))
{
	// left key is pressed: move our character
	this->move(-pSpeed, 0.f);
}
boundingBox.left += pSpeed;
// check ahead in direction
boundingBox.width += pSpeed;
```