Report for 2D Platform Game

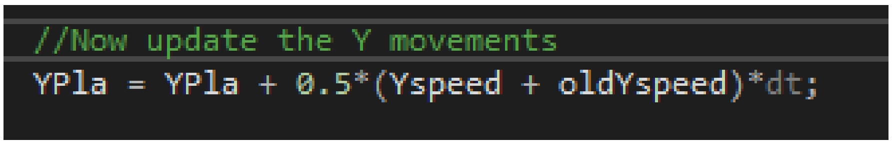
1. Introduction

To control the player's movements in a 2D platform game, several considerations and careful planning has had to be carried out to make the game as lifelike as possible. In this report we will be discussing the techniques used for simulating the player's movements as well as how collisions are detected between the player and the platforms and finally we will finish with a conclusion.

2. Player Movements

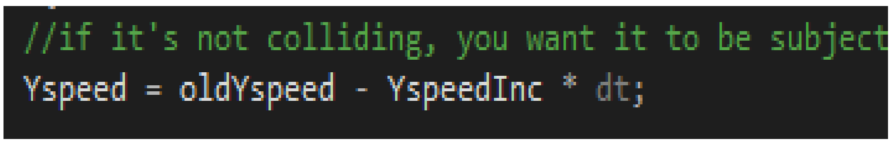
2.1 Jumping and Moving from Left to Right

When the user wants the player character to jump, the initial step is to set its speed to 70. The player character also has a jump pressed flag which is initially set to true when the user presses jump. This is only set to false when the player character lands on top of a platform (without this, the player would be able to keep jumping multiple times in the air). On each frame load, the formula displayed below is called. It simply updates the Y position with the new calculated Yspeed (oldYspeed is set to zero when the player jumps).



```
//Now update the Y movements
YPla = YPla + 0.5*(Yspeed + oldYspeed)*dt;
```

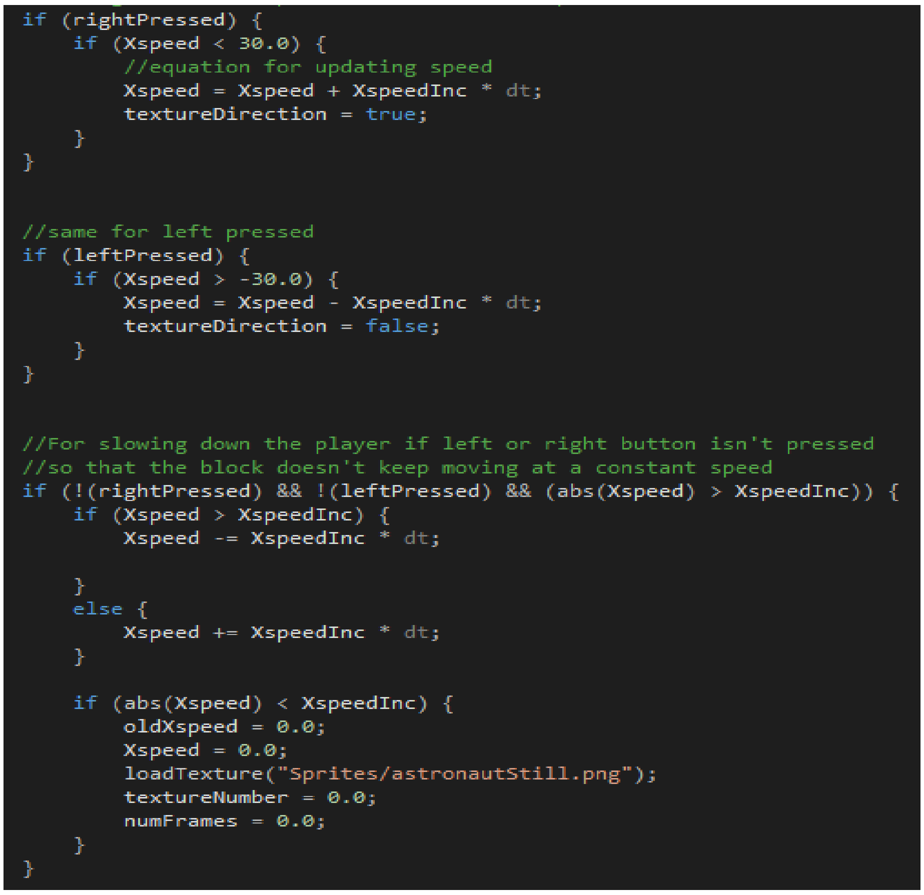
The next step is to update its Yspeed. As the player is subject to gravity in the game, it needs to decrease its Yspeed and this simulates a falling motion. The formula used for this is displayed below.



```
//if it's not colliding, you want it to be subject
Yspeed = oldYspeed - YspeedInc * dt;
```

It simply decrements the Yspeed with a constant factor known as YspeedInc. This formula is called only if the player isn't colliding with anything. When the player hits the top of the platform, its Yspeed is set to zero and the jump pressed flag is set to false. The important thing to note is the dt appended to the ends of both formulas. This is a constant time increment to ensure that the speed of the game is platform independent.

For updating X position and Xspeed, a similar process is used. Firstly it checks whether the right or left keys have been pressed. If so, it increments Xspeed accordingly (increases if right or decreases if left) up to a max value (in this case 30). If they are not held down, the Xspeed is decremented in the opposite direction to which the player is moving. It does this continuously until the player character stops moving.

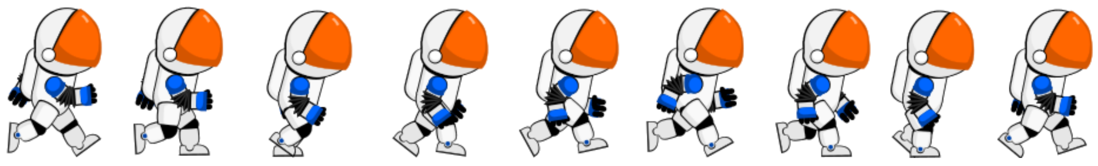


```
if (rightPressed) {
    if (Xspeed < 30.0) {
        //equation for updating speed
        Xspeed = Xspeed + XspeedInc * dt;
        textureDirection = true;
    }
}


//same for left pressed
if (leftPressed) {
    if (Xspeed > -30.0) {
        Xspeed = Xspeed - XspeedInc * dt;
        textureDirection = false;
    }
}


//For slowing down the player if left or right button isn't pressed
//so that the block doesn't keep moving at a constant speed
if (!(rightPressed) && !(leftPressed) && (abs(Xspeed) > XspeedInc)) {
    if (Xspeed > XspeedInc) {
        Xspeed -= XspeedInc * dt;

    }
    else {
        Xspeed += XspeedInc * dt;
    }

    if (abs(Xspeed) < XspeedInc) {
        oldXspeed = 0.0;
        Xspeed = 0.0;
        loadTexture("Sprites/astronautStill.png");
        textureNumber = 0.0;
        numFrames = 0.0;
    }
}
```

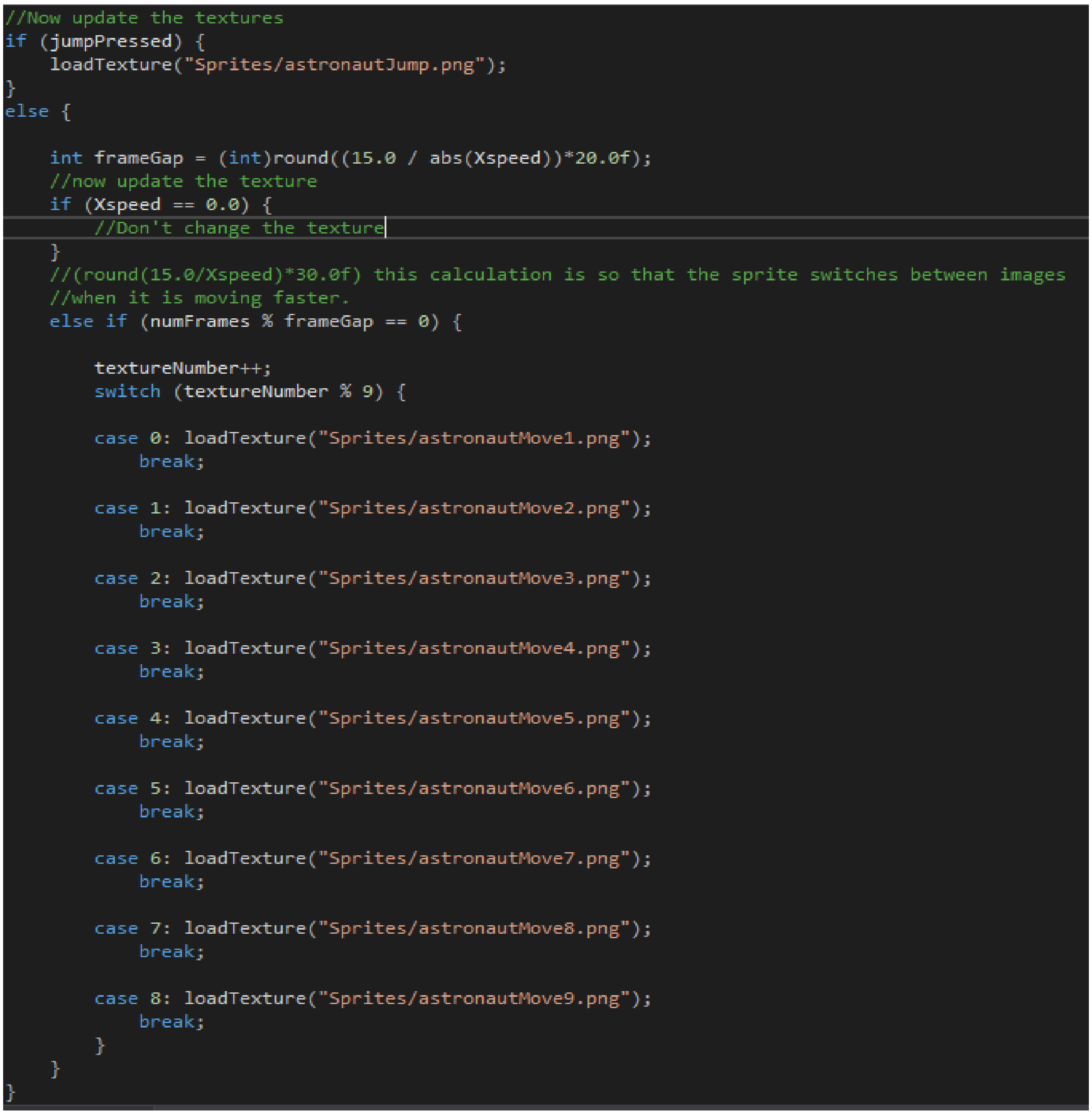
2.2 Animation of Player Movements

For animating the player whenever it jumps or moves, a simple sprite sheet was used. Displayed below are the textures used to represent the player character. The left image is when the character is stationary while the right image is when it is jumping or in the air.

For jumping, the process is simple. If the jump pressed flag has been set, then load the jumping image. Otherwise, load the still image. For animating when the character is moving, it uses 9 different images that are loaded continuously (when the player is moving).



It switches between each image depending on the speed it is moving. The frameGap calculation is used to decide when to switch an image and is dependent on the character's Xspeed. If the current numFrames (a global variable for recording the number of frames rendered) is a multiple of frameGap it changes the image of the player character. When it stops moving, it loads the still image as before.



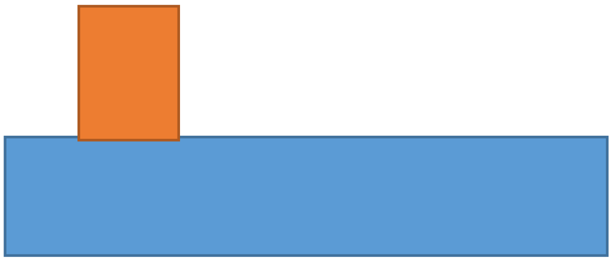
```
//Now update the textures
if (jumpPressed) {
    loadTexture("Sprites/astronautJump.png");
}
else {

    int frameGap = (int)round((15.0 / abs(Xspeed))*20.0f);
    //now update the texture
    if (Xspeed == 0.0) {
        //Don't change the texture
    }
    //(round(15.0/Xspeed)*30.0f) this calculation is so that the sprite switches between images
    //when it is moving faster.
    else if (numFrames % frameGap == 0) {

        textureNumber++;
        switch (textureNumber % 9) {

        case 0: loadTexture("Sprites/astronautMove1.png");
            break;

        case 1: loadTexture("Sprites/astronautMove2.png");
            break;

        case 2: loadTexture("Sprites/astronautMove3.png");
            break;

        case 3: loadTexture("Sprites/astronautMove4.png");
            break;

        case 4: loadTexture("Sprites/astronautMove5.png");
            break;

        case 5: loadTexture("Sprites/astronautMove6.png");
            break;

        case 6: loadTexture("Sprites/astronautMove7.png");
            break;

        case 7: loadTexture("Sprites/astronautMove8.png");
            break;

        case 8: loadTexture("Sprites/astronautMove9.png");
            break;
        }
    }
}
```

3. Collisions with Platforms

For detecting whether a collision has occurred at all, the player uses an OBB (Oriented bounding boxes) to do this. This works by projecting an axis that is perpendicular to an edge of the polygon that you want to test. Then you project the vertices of each polygon onto this axis and check for any overlap. If there is an overlap then a collision has occurred.

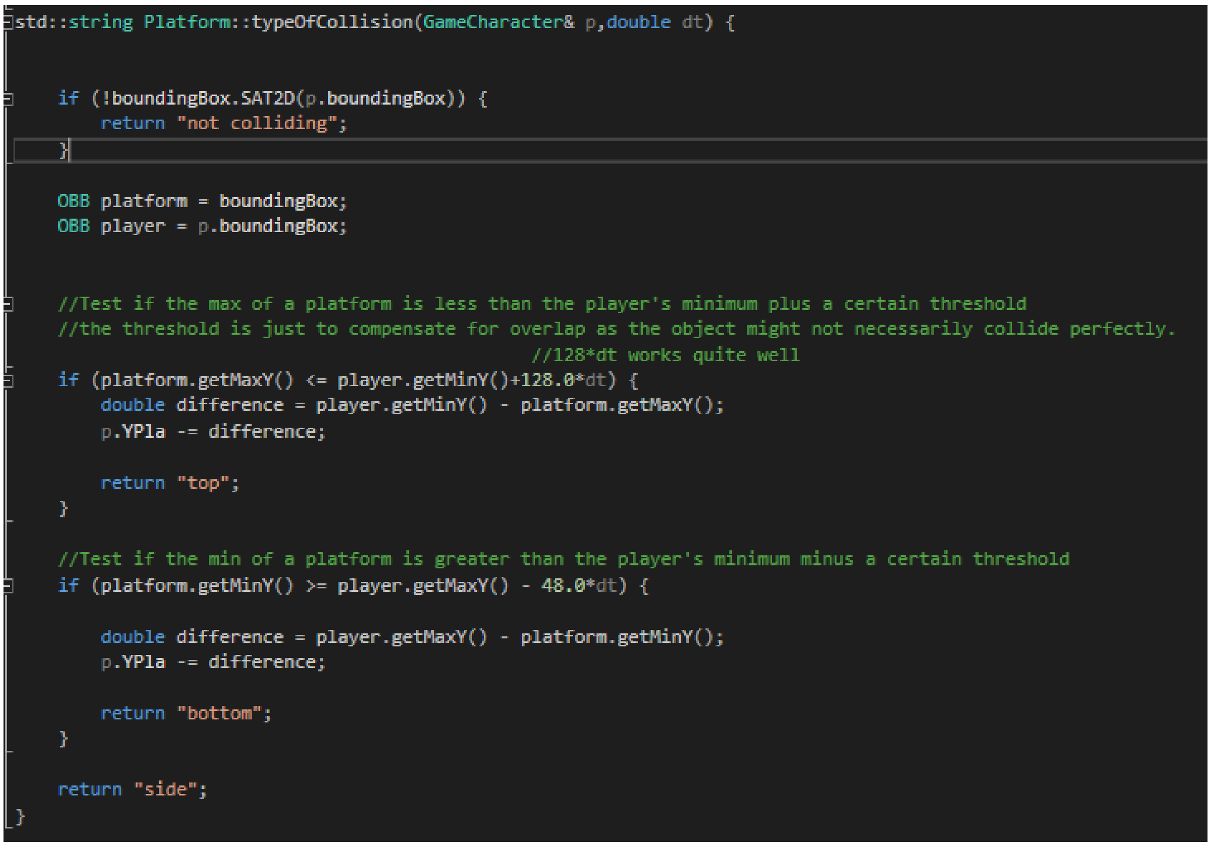
3.1 Colliding from Different Directions

If the OBB has found a collision, further processing is required to calculate the side of the platform that the player has collided with. The platform has a type of collision method that returns a string representing the side of the platform that the player has collided with. For a top collision, the test is whether the MaxY of the platform is less than or equal to the MinY of the player plus a certain threshold (this is to compensate for overlap as it is very unlikely for the object to collide perfectly).



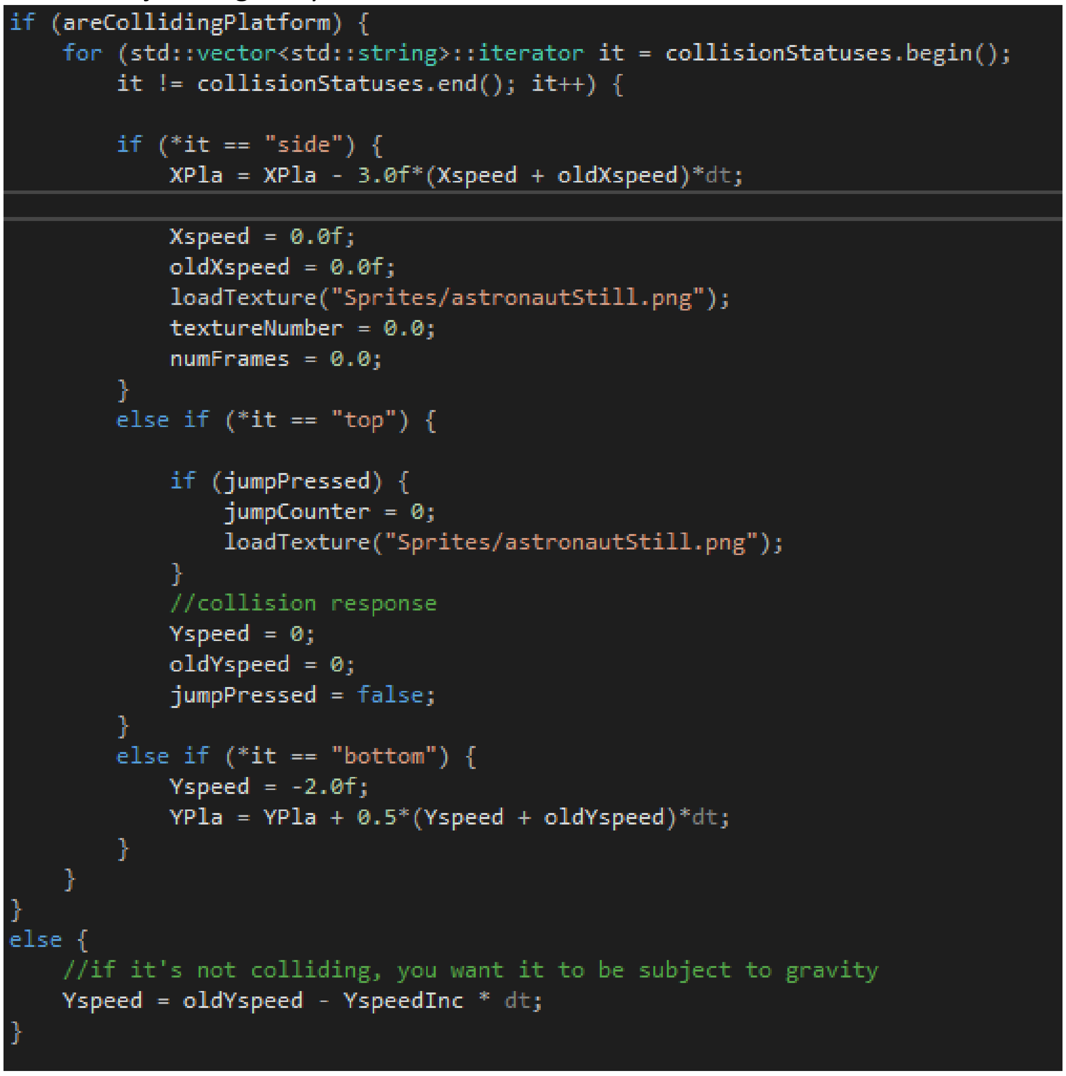
An example of a top collision. Notice how the MinY of the player is slightly less than the MaxY of the platform. This indicates a top collision

The test for a bottom collision is whether the MinY of the platform is greater than or equal to the MaxY of the player minus a certain threshold (this threshold is less for a bottom collision because the player's speed will be much less than for a top collision so less overlap occurs). If the collision doesn't pass any of these tests, then the player has collided with the side of the platform.



```
std::string Platform::typeOfCollision(GameCharacter& p,double dt) {


    if (!boundingBox.SAT2D(p.boundingBox)) {
        return "not colliding";
    }

    OBB platform = boundingBox;
    OBB player = p.boundingBox;


    //Test if the max of a platform is less than the player's minimum plus a certain threshold
    //the threshold is just to compensate for overlap as the object might not necessarily collide perfectly.
                                            //128*dt works quite well
    if (platform.getMaxY() <= player.getMinY()+128.0*dt) {
        double difference = player.getMinY() - platform.getMaxY();
        p.YPla -= difference;

        return "top";
    }

    //Test if the min of a platform is greater than the player's minimum minus a certain threshold
    if (platform.getMinY() >= player.getMaxY() - 48.0*dt) {

        double difference = player.getMaxY() - platform.getMinY();
        p.YPla -= difference;

        return "bottom";
    }

    return "side";
}
```

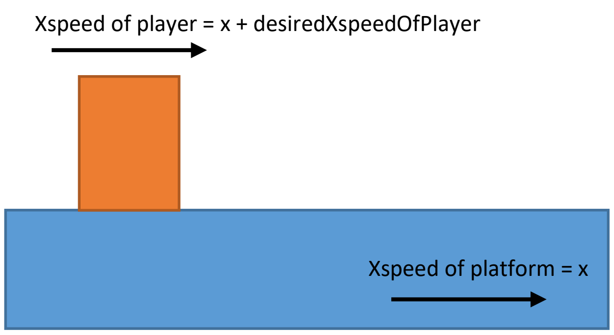
Collision response is handled in the player character class. When the platform returns a collision status, it is pushed into a vector data structure that is stored in a player character object which in turn the player iterates through this vector and responds accordingly. For a side collision, its Xspeed and oldXspeed are both set to zero and likewise for a top collision its Yspeed and oldYspeed are both set to zero. For a bottom collision, its Yspeed is set to -2 is then subject to gravity.

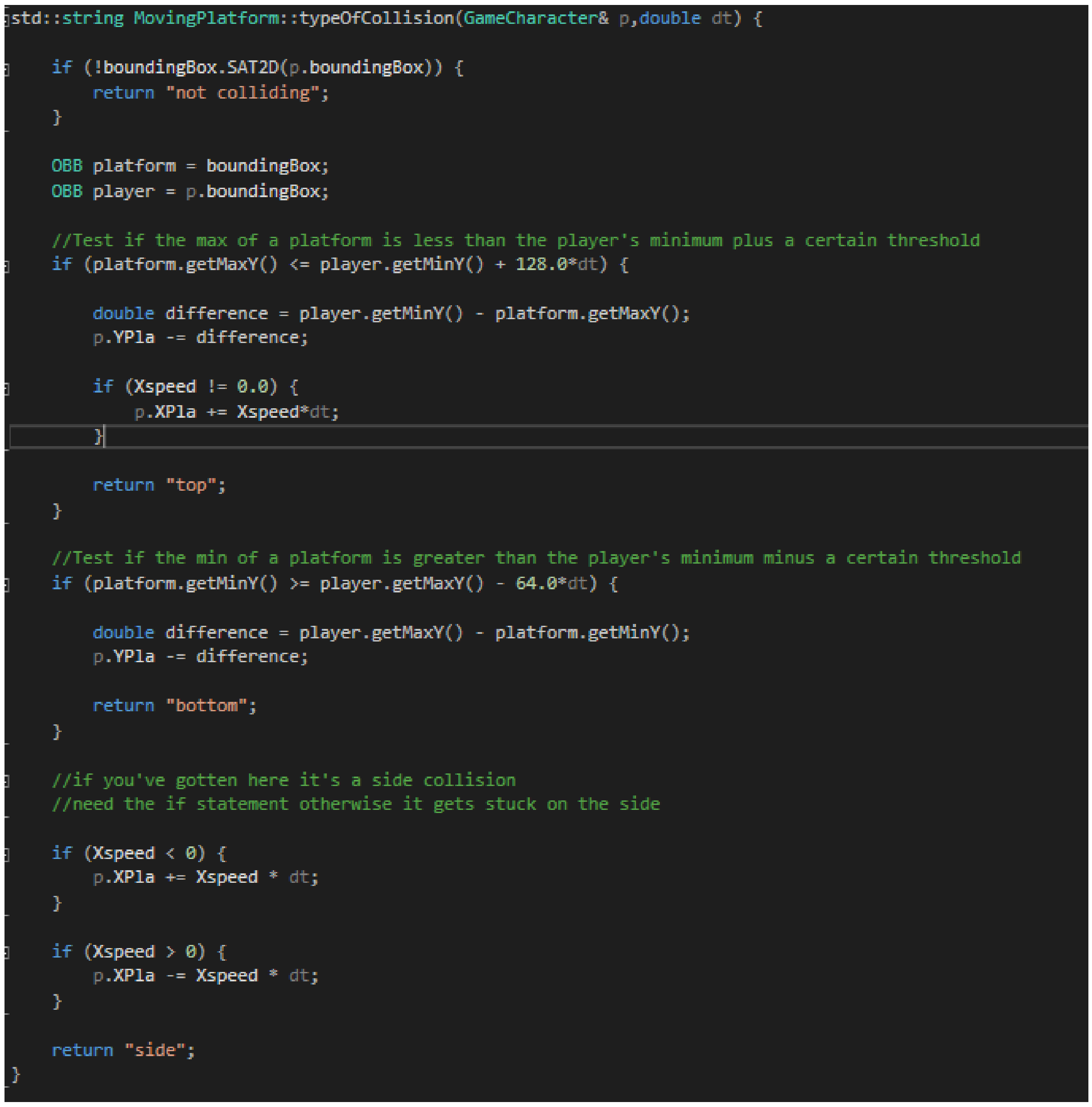


```
if (areCollidingPlatform) {
    for (std::vector<std::string>::iterator it = collisionStatuses.begin();
        it != collisionStatuses.end(); it++) {

        if (*it == "side") {
            XPla = XPla - 3.0f*(Xspeed + oldXspeed)*dt;

            Xspeed = 0.0f;
            oldXspeed = 0.0f;
            loadTexture("Sprites/astronautStill.png");
            textureNumber = 0.0;
            numFrames = 0.0;
        }
        else if (*it == "top") {

            if (jumpPressed) {
                jumpCounter = 0;
                loadTexture("Sprites/astronautStill.png");
            }
            //collision response
            Yspeed = 0;
            oldYspeed = 0;
            jumpPressed = false;
        }
        else if (*it == "bottom") {
            Yspeed = -2.0f;
            YPla = YPla + 0.5*(Yspeed + oldYspeed)*dt;
        }
    }
}
else {
    //if it's not colliding, you want it to be subject to gravity
    Yspeed = oldYspeed - YspeedInc * dt;
}
```

3.2 Transitional Platforms

For transitional platforms additional collision responses were required which depended on where the collision occurred. If the player has collided on top of the platform, its X position is incremented by the moving platform's X speed. This is to ensure that the player character is moving with the platform. For a bottom collision the response doesn't change. For a side collision, the X position of the player character is always incremented with the opposite direction of the moving platform. This is to avoid the player character getting stuck at the side of the platform. As before, the method returns a string and is dealt with in the player character class.





```
std::string MovingPlatform::typeOfCollision(GameCharacter& p,double dt) {

    if (!boundingBox.SAT2D(p.boundingBox)) {
        return "not colliding";
    }

    OBB platform = boundingBox;
    OBB player = p.boundingBox;

    //Test if the max of a platform is less than the player's minimum plus a certain threshold
    if (platform.getMaxY() <= player.getMinY() + 128.0*dt) {

        double difference = player.getMinY() - platform.getMaxY();
        p.YPla -= difference;

        if (Xspeed != 0.0) {
            p.XPla += Xspeed*dt;
        }

        return "top";
    }

    //Test if the min of a platform is greater than the player's minimum minus a certain threshold
    if (platform.getMinY() >= player.getMaxY() - 64.0*dt) {

        double difference = player.getMaxY() - platform.getMinY();
        p.YPla -= difference;

        return "bottom";
    }

    //if you've gotten here it's a side collision
    //need the if statement otherwise it gets stuck on the side

    if (Xspeed < 0) {
        p.XPla += Xspeed * dt;
    }

    if (Xspeed > 0) {
        p.XPla -= Xspeed * dt;
    }

    return "side";
}
```

4. Conclusion

In conclusion, there were many techniques that were used to simulate the movements and collision responses within the 2D platform game. These were used to make the character able to collide with platforms from multiple directions, have animated movement and produce horizontal transitional platforms. Further work could be implementing vertically transitional platforms and eliminating the overlap between the player character and platforms.