Computer Graphics

Aum Patel - 180000492

May 6, 2021

Overview

The project I have created is based around a 'candy cane' style theme, where everything is vibrant and multi-coloured.

Annotated Game Scene

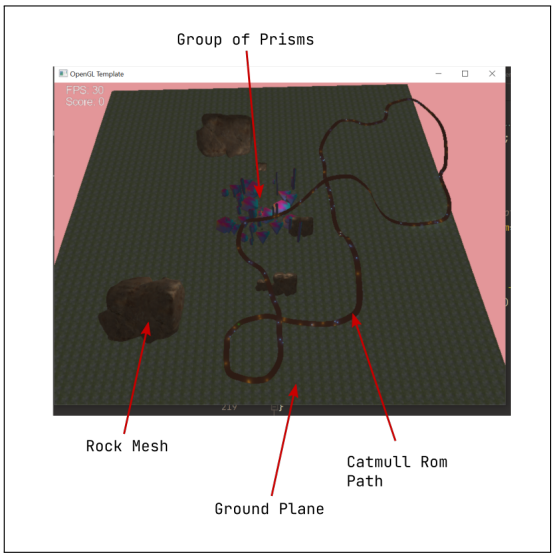

Figure 1: Top Down view of the game scene, zoomed out

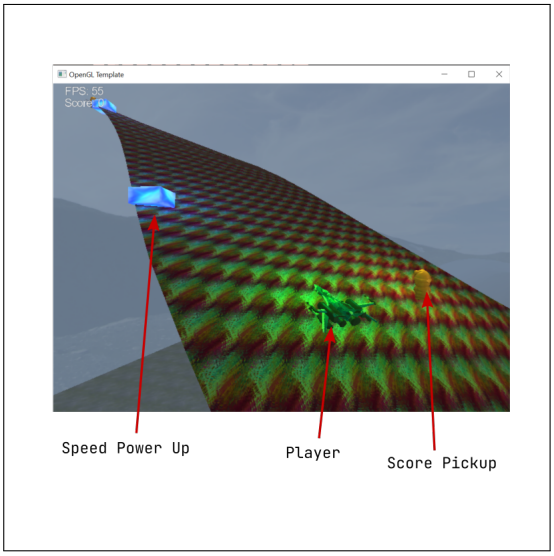

Figure 2: Close view of the player, speed power up and score power up

User Guide

The goal of the game is to collect as many chocolate santa clause's as possible to get a high score. The player can pick up the blue pointed shape to gain a temporary speed boost.

Controls	Description
Up Arrow	Accelerate
Down Arrow	Deccelerate
Left Arrow	Move Left
Right Arrow	Move Right
F	Switch Camera View
W A S D	Camera Movement in Free Roam

Table 1: Controls for the game

Route and Camera

Route

To plan out the path, Blender was used, as this provided an easier time visualising it over pen and paper. After sketching, I measured and listed the vertices out which are then used in the Catmull

Rom `SetControlPoints()` method; the measurement displayed in the image below are rounded to the nearest fifth, as the Blender measure tool does not snap.

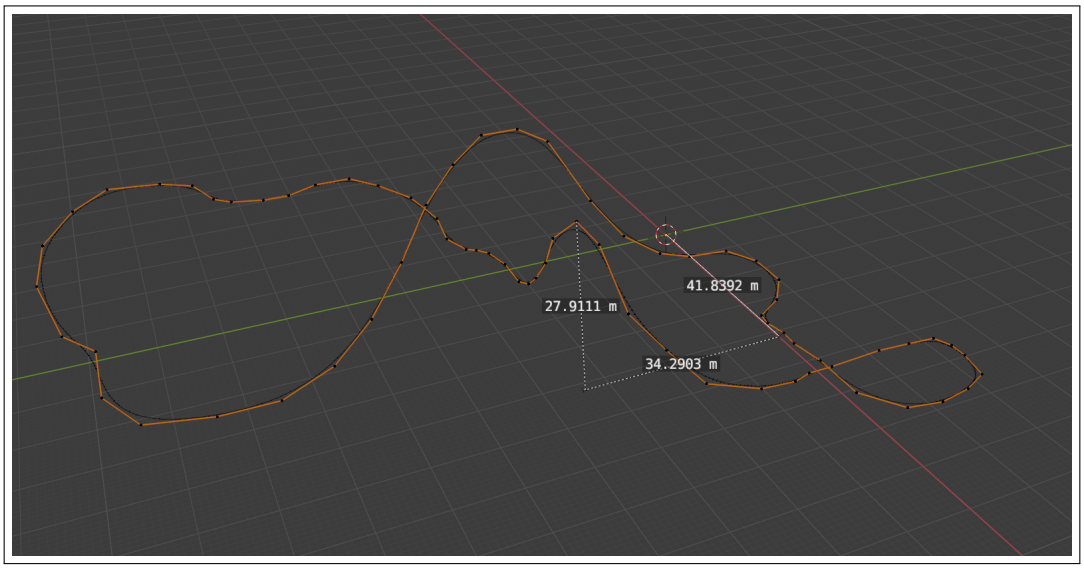

Figure 3: Path Sketched out in Blender

When adding the points into the `SetControlPoints()` method, I decided to create some parameters and add/multiply those to the vertices I was adding in, this way I could move the path around freely without needing to adjust the vertices.

```
SetControlPoints(){
    // parameters to adjust the scaling and positioning of the path.
    const float xScale = 10.f;
    const float zScale = 10.f;
    const float yScale = 10.f;
    const float xOffset = 0 * xScale;
    const float zOffset = 0 * zScale;
    const float yOffset = 10 * yScale;

    // example for one control point. Total of 70 points
    m_controlPoints.emplace_back(xOffset + xScale * 59.0, yOffset + yScale * 5.0,
        ↪ zOffset + zScale * 7.0);
}
```

Camera

I have implemented 5 camera modes: **First Person**, **Third Person**, **Left Side View**, **Right Side View** and **Top View**.

I switch between them by storing the current `cameraType` as an integer, and have a set of camera modes:

```
    static const glm::uint FIRST_PERSON = 0;
    static const glm::uint THIRD_PERSON = 1;
    ... // and so on
```

When switching camera mode, I change the camera type and check against this set of constant integers when setting the view of the camera by the spline and place the camera relative to the path with different distance values.

I generate a TNB frame in every instance and offset the camera's position relative to the TNB frame to move it along local coordinates. For the **first person view** I moved it up one value on the bi-normal to place it just above the nose of the spaceship.

For the **3rd person view**, I moved it even higher, and placed it slightly behind the car so the whole track can be seen. When in 3rd person the camera does not follow the left and right movement of the car like it does in First Person.

For the **side views** I multiply an positive/negative (for both left/right sides) offset value by the normal and place the camera's 'look at' at the car's position.

Similar with the **top view**, moving it up on the binormal and having the camera look down at the car.

As well as these views, I still have the free view available to toggle to making it easy to debug.

HUD, Gameplay and Advanced Rendering

HUD

To create the HUD, I chose to use full-screen PNG's that act as sprites in the game and are rendered on top of the rendered frame using a full-screen quad. I created a `HudItem` class which holds one or multiple textures and can be 'bound' to the renderer by passing in an index; these textures are stored inside a vector. The `HudManager` will then render the many `HudItems` that are created.

```
void HudManager::Create() {
    ...
    // create a full screen quad
    ...
    santa.AddTexture("resources\\textures\\hud\\santa_png\\santa.png");

    for (int i = 1; i <= 6; ++i) {
        speedometer.AddTexture("resources\\textures\\hud\\speedometer_png\\speed_"
            ↪ + to_string(i) + ".png");
    }
    speedometer.AddTexture("resources\\textures\\hud\\speedometer_png\\speed_boost.
        ↪ png");
}

void HudManager::Render(CShaderProgram* shader) {
    // Draw santa
    glBindVertexArray(quadVAO);
    santa.Bind(0);
    glDrawArrays(GL_TRIANGLES, 0, 6);

    // Draw Speedometer
    glBindVertexArray(quadVAO);
    // will draw the appropriate speedometer texture
    speedometer.Bind(speedometerIndex);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

Below are figures that show how the different hud items are displayed.

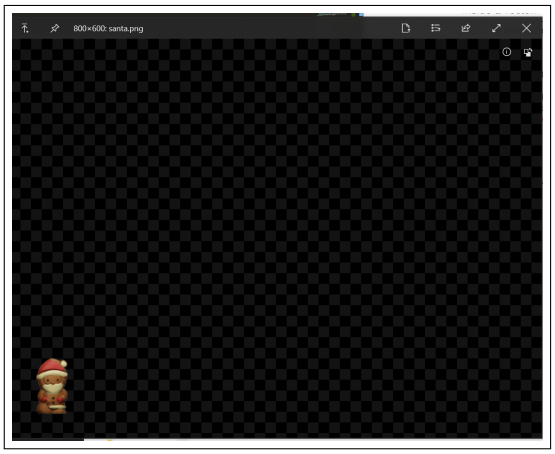

Figure 4: Score HUD Item

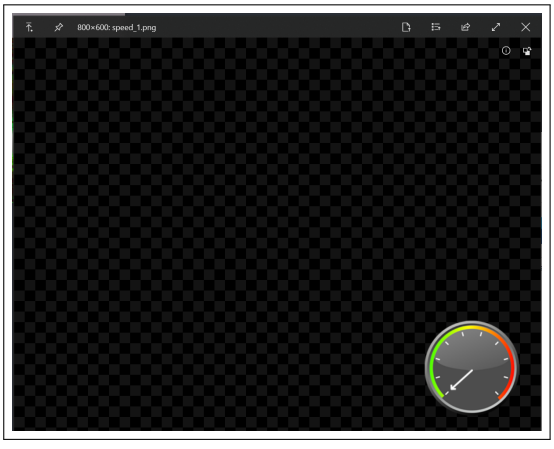

Figure 5: Speedometer Hud Item

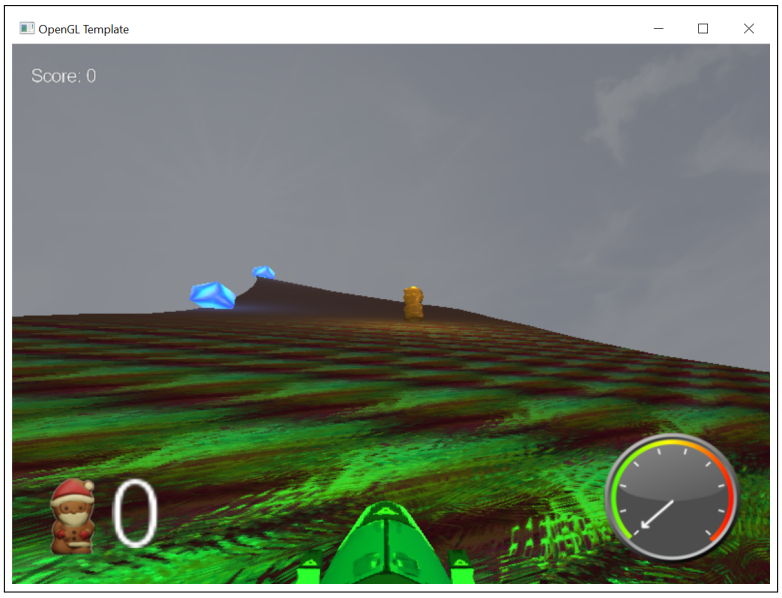

Figure 6: Hud as Displayed in Game

As well as the two HUD classes that I created, I also needed a shader to render the 2D elements appropriately. This is a very basic shader that maps the texture's colour to the fragment shader. This is all then rendered after the radial blur in the main Game `Render()` method.

Gameplay

Control

By pressing the upwards arrow key, the car will accelerate until it reaches the max limit, and vice versa when pressing the down arrow key; when the key is release, the speed will be kept unchanged. This is done by creating a speed value and setting it in both the **Car** and **Camera** objects, and changing the value that is being set on the appropriate key presses.

When pressing the Left and Right Arrow keys, the car placed on an offset from the center of the Catmull Rom path. This offset is applied via the TNB frame on the Normal of the car and the camera (only applied to the camera when in First Person View).

```
    mPlayerXOffset -= m_dt * mCar->getXOffsetSpeed(); // + for moving right
    mCar->setXOffset(mPlayerXOffset);
    m_pCamera->setXOffset(mCar->getXOffset());
```

Collision Detection

The collision detection used in the game is very simple, however is made easier by using a generalised `GameObject` class, this class contains the collision radius of an object. Any object that is rendered on screen is a child of GameObject. Every frame in the `Update()` loop of my game, I call a function `ManageCollisions()`, which traverses the many vector's that store the positions to render the different game objects and calls `CheckCollision(aPos, aRadius, bPos, bRadius)` that compares the radius between two objects and returns true if they overlap.

Depending on which object it collides with, a different behaviour occurs. For the score pickup, the player's score is incremented by 1, and if it is te speed power up, then `mSpeedPowerUpTimer` is set to 1 seconds, which makes a boost activate.

When a collision happens, I store the iterator/position inside of the vector of positions and remove it; On doing so, the object in that position will stop rendering, simulating the player picking it up.