Computer Graphics

Aum Patel - 180000492

May 7, 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

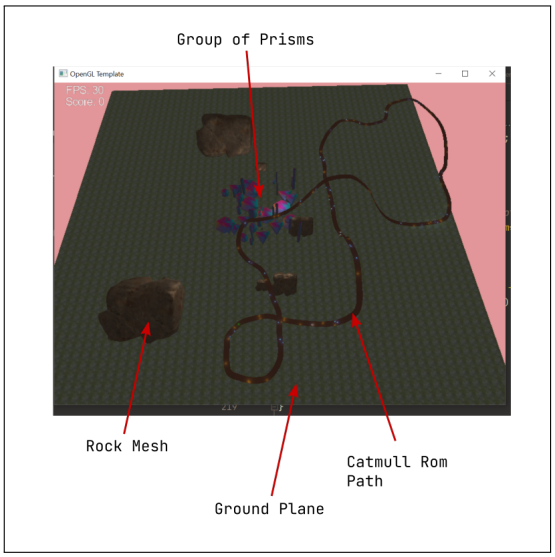

Figure 1: Top Down view of the game scene, zoomed out

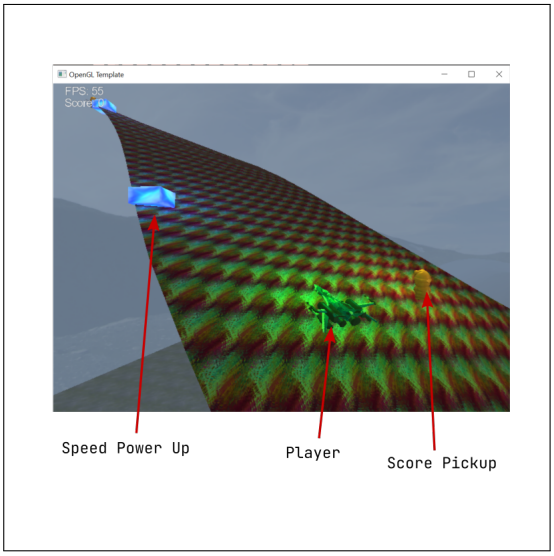

Figure 2: Close view of the player, speed power up and score power up

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.

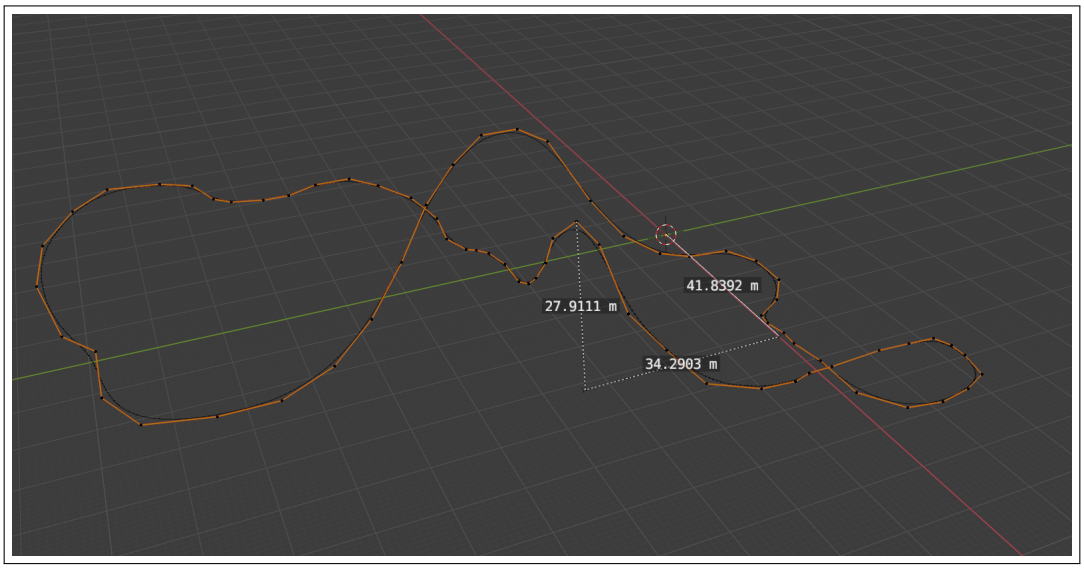

Figure 3: Path Sketched out in Blender

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, these parameters were a scalar value on the xyz axis and an offset value on the xyz (which is multiplied by the scalar before being applied to the control points). This way I could move the path around freely without needing to adjust the vertices.

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 to the texture wanting rendered; these textures are stored inside a vector. The `HudManager` will then render the many HudItems that are created.

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

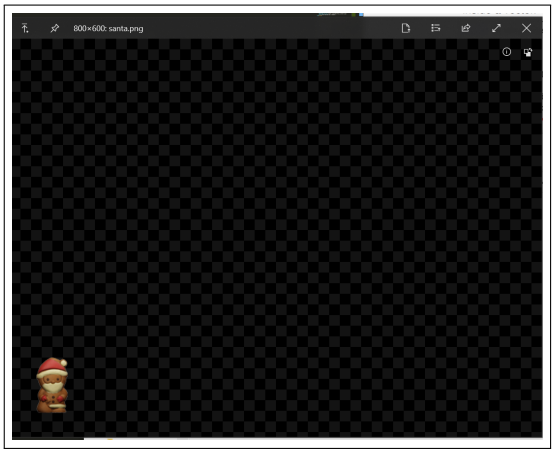

Figure 4: Score HUD Item

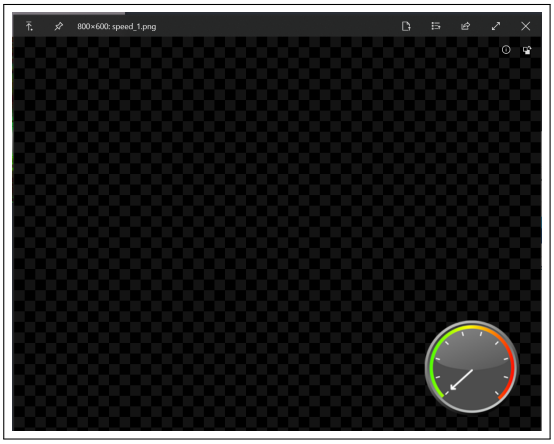

Figure 5: Speedometer Hud Item

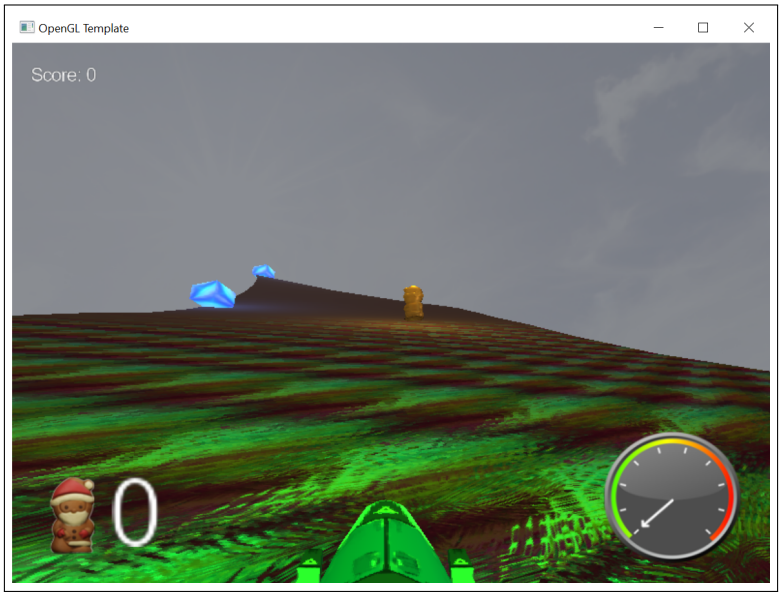

Figure 6: Hud as Displayed in Game

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. This shader was created by taking the radial blur shader, which was already a 2d shader, and removing the values that created the blur effect, essentially giving me a simple 2d shader.

When rendering multiple full-screen elements, the `GL_DEPTH_TEST` had to be disabled, otherwise only the first element rendered will show; disabling depth test is done to avoid 'plane-fighting' where two

overlapping faces will clash with each other, however, for the HUD, rendering multiple quads on the same 'plane' needed this to be disabled, to have all elements rendered.

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.

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.

Gameplay Elements

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 the speed power up, then `mSpeedPowerUpTimer` is set to 1 seconds, which makes a boost activate. The player just needs to pickup as much chocolate as they can. If they pick up all the points, a game over message will display.

Figure 7: Game Over Screen

Advanced Rendering

Fog

Two advanced rendering techniques were implemented, the first on was fog. This was achived by following a tutorial [1]. This tutorial described the shader that had to be created, I followed this and adjusted the parameters to get fog that displayed correctly; the `mainShader.vert/frag` was changed to add the fog. As the distance from the player to the object is calculated in the vertex shader per vertex, the distance for large objects, such as the terrain and skybox, is not presented correctly. To rectify this, the density and gradient for both gradient and skybox was lowered to make them visible.

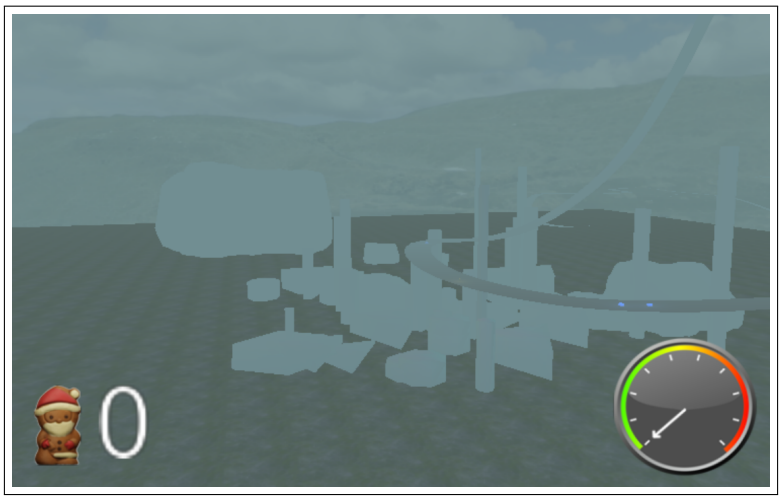

Figure 8: Fog in game scene

To add to the 'candyland' style theme of the game, the colour of the fog was randomly adjusted on the client side, and passed through to fragment shader as the sky colour, giving a nice vibrant ambience.

Radial Blur

The radial blur was achieved by creating a seperate shader files (`radialBlur.vert/frag`) for it. These shaders were created by following the tutorial provided in lecture 7 [2]. As this tutorial/post was 10 years old, the shader code had to be ported over to version 400. The fragment shader takes 10 samples

[1] Tutorial by Thin Matrix for Fog in OpenGL `https://youtu.be/qslBNLeSPUc`

[2] INM376_2021.L07.pdf, page 55 `https://stackoverflow.com/questions/4579020/how-do-i-use-a-glsl-shader-to-apply-a-radial-blur-to-an-entire-scene`