Creating a 3D Racing Game in OpenGL

By Noah Harper

Computer Science Department

Cal Poly, San Luis Obispo

June 16, 2015

1 Introduction

For my Senior Project, I set out to make a three-dimensional racing game, with inspiration from old Nintendo classics such as Mario Kart and F-Zero. The idea was to create a world rendered in OpenGL and to design simple mechanics that would allow for a fun and straight-forward gameplay experience. Making a fully functioning three-dimensional driving game with playable physics proved to be a much more challenging tasks than it originally appeared. I spent roughly half of my time building geometry and implementing graphics technologies in OpenGL, and the other half toiling with rigid body dynamics and collisions.

I began coding my game with two-dimensional physics and found this to be relatively simple, which gave me a false sense of security. When I moved my game into three dimensions, the rigid body physics and the collision handling got significantly more complex. This proved to be a major challenge to overcome. In the end, I achieved success in some areas and fell short in others.

The rendering portion of my code turned out to be successful with a number of technologies implemented including textures, shadows, fog, camera turn lag, multi-sampling (unsupported on the computer used in the screenshots below), and Blinn-Phong shading.

The following sections will go over the major technologies I implemented and talk about the general approaches I took and the challenges I faced.

2 Spline Roads

A good racing game needs roads. Simple sections of roads may be represented with planes and cubes. Some sections may even be implied by cluttering geometry on the sides of the road, effectively restricting the player to a roadlike area. However, to have a true race track, it is necessary to have smooth, and potentially banked curves. To accomplish this, I looked into using spline curves.

The idea behind building a track with spline curves is to use the turning curving spline to represent the center of the track. From there it is a matter of extruding points outward from the spline and connecting those points to create triangles. The end result is a mesh that resembles a road. Figure 1 shows the road curving between two spheres.

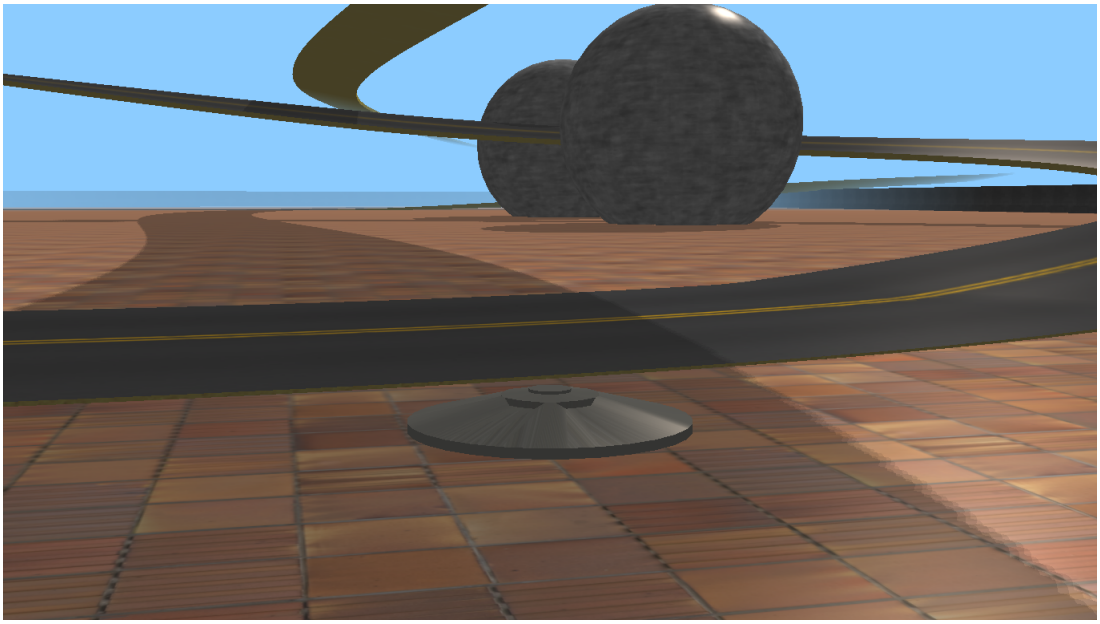

Figure 1: Winding road.

There were, however, a number of challenges I faced in being able to properly extrude the side points of the track. First, I had to use arc length parameterization in order to acquire evenly spaced points so that the mesh would have triangles of a consistent size. From there, I needed a vector that would represent the "up" direction of the road. For example, this vector was usually set to (0, 1, 0), making the flat top of the road face towards the sky. A different up direction could be used to represent a banked curve. This up vector was then used to find the side vectors needed to

extrude the points representing the edges of the road. This was done by taking the cross product of the up vector and the forward vector (the vector between the current point on the curve and the next point along the curve). The triangles can be seen in Figure 2. Finally, I needed to find the normal vectors at each point so the curve could be shaded properly. This was done with another cross product, only this time between the newly acquired side vector and the forward vector.

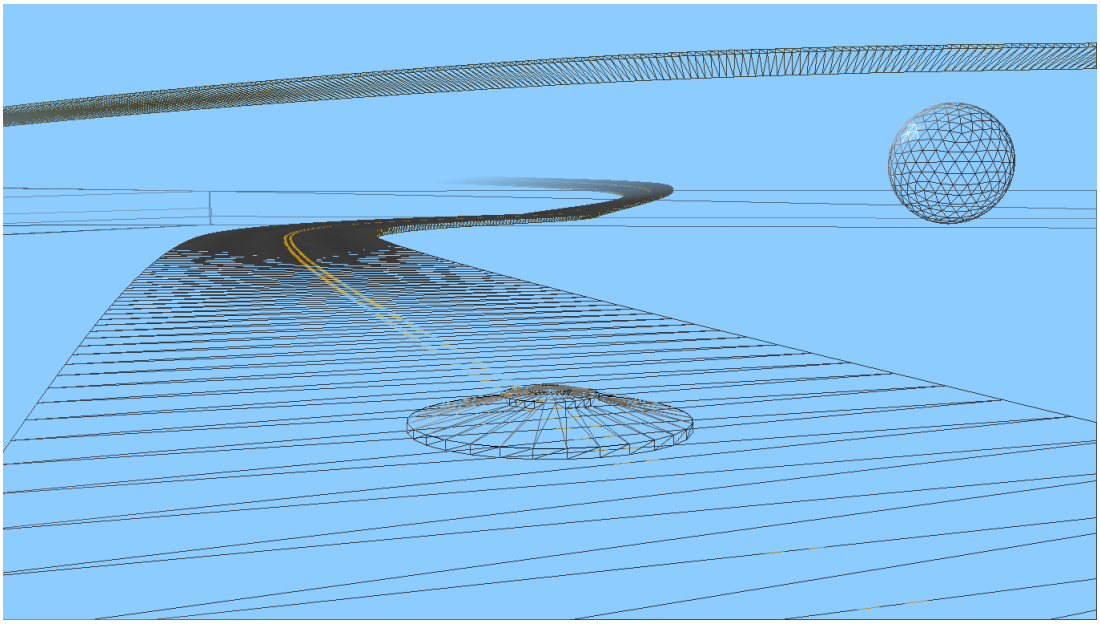

Figure 2: The road's geometry.

I then added dimension to the mesh by copying the points and translating them downwards, then building triangles to represent the sides and bottom of the three-dimensional road, as seen in Figure 3. I stored the sides and bottom in a new mesh so they could be textured separately from the top of the road. I textured the top of the road with a repeating road pattern, while the sides and the bottom had a simple concrete texture. In order to make the texturing repeat correctly I had to find the right texture coordinates. I did this by splitting the road into shorter segments of ten planes and ordered the texture coordinates to increase along each of the planes, demonstrated in Figure 4. With the road texturing properly, my mesh creation was complete.

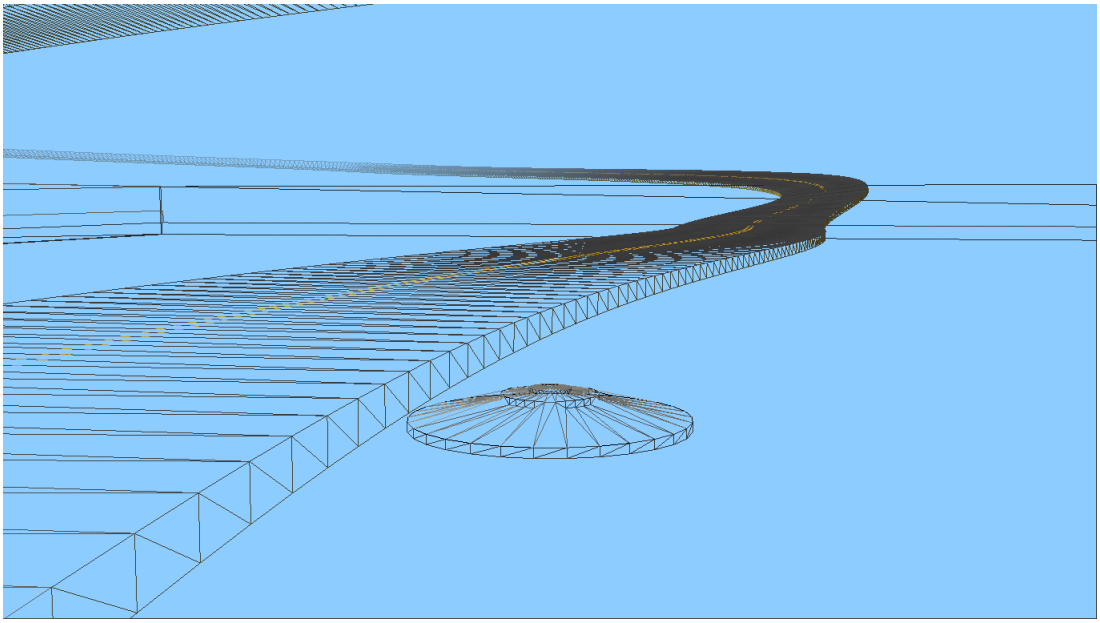

Figure 3: The road's 3D geometry.

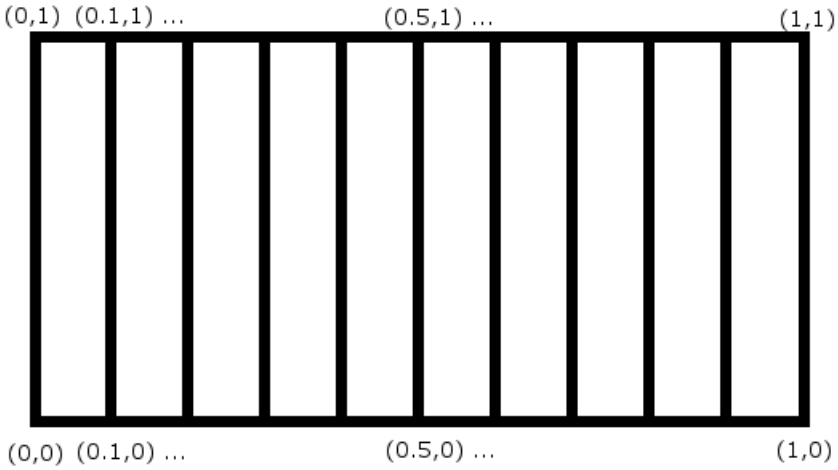

Figure 4: Texture coordinates.

3 Graphics

3.1 Shadows

The most substantial graphics technology that I implemented for my game was rendering shadows using the shadow mapping technique. Shadows are key to establishing relationships between different objects by displaying the way one object blocks light from hitting another. Adding shadows to my scene helped give the world a much better sense of depth. The shadows can be seen in Figure 5 and Figure 6.

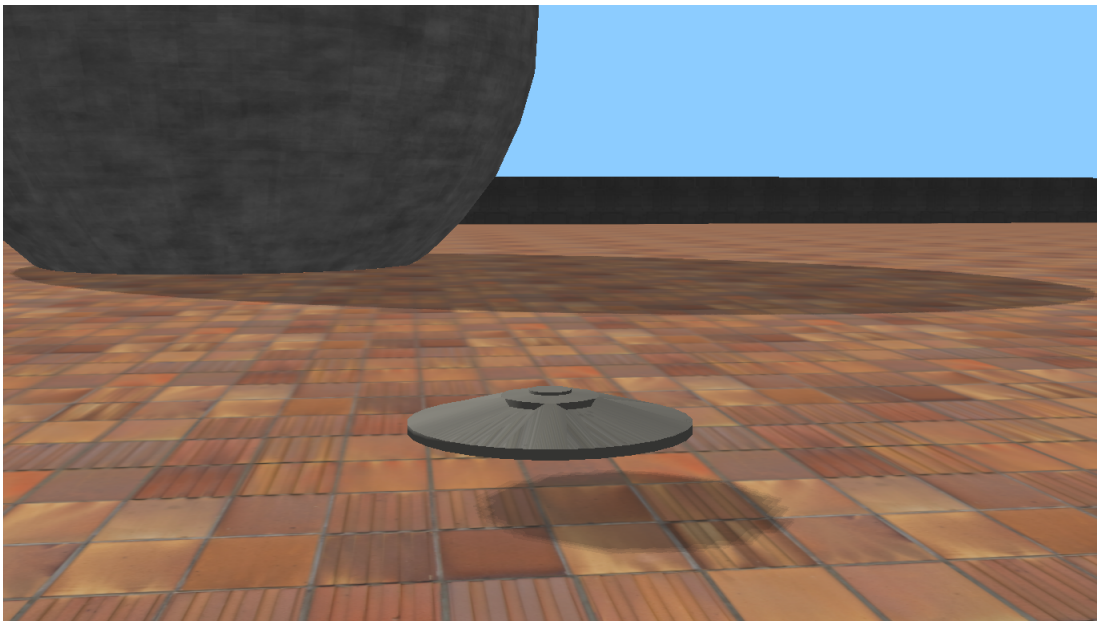

Figure 5: A sphere casting a shadow.

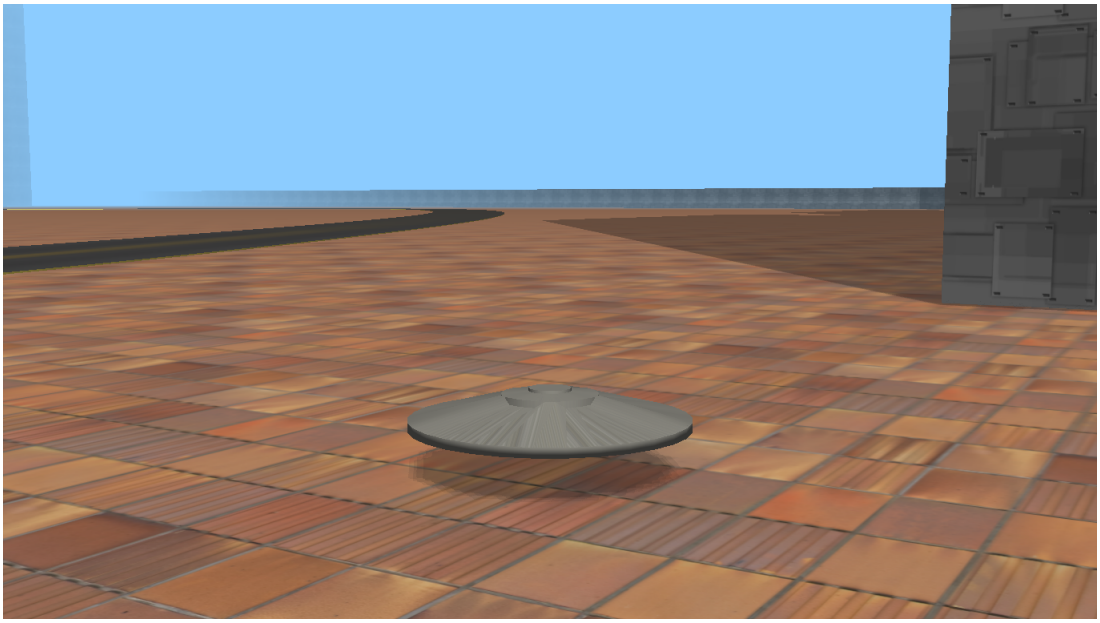

Figure 6: A building casting a shadow.

The algorithm I used for shadow mapping is accomplished through a two pass rendering method. In the first pass, the scene is rendered from the perspective of the light source. However, instead of rendering the scene as normal, this pass only renders the depth of the geometry seen. This depth must be stored in a buffer object that is created beforehand. Next, in the second pass, the scene is rendered from the cameras perspective. This time, the geometry is shaded as usual with the added step of comparing each fragments distance from the light source to the depth stored in the depth buffer. If the distance to the light source is further than the distance stored in the buffer, then the fragment is in the shadow and should be shaded darker. Figure 7 illustrates the scene and the distances being compared.

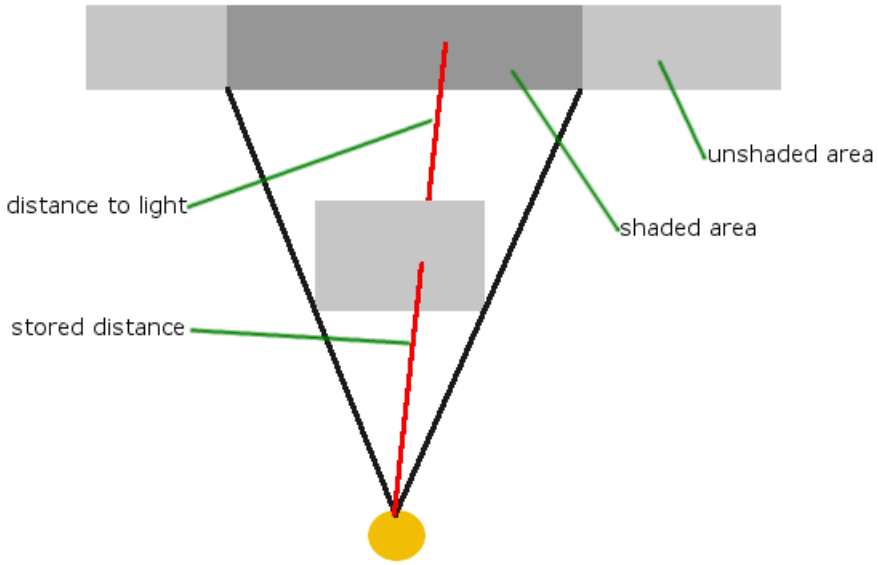

Figure 7: Depth comparison for shadowed region.

One limitation with this method is that the quality of the shadows suffer when too much of the scene is shadowed at once. To combat this, I determined a point that is a set distance in front of the player. The lights perspective is then set to look at this point and the lights field of view is tightened. This way, the light "sees" less, causing less of the scene to be shadowed, which improves the resolution of the depth buffer. And, since it is set to look in front of the player, the player always sees the relevant shadows. This way, even with only a four sample filter, the shadows look decently sharp.

3.2 Fog

Although a much smaller technology than shadows, I still thought it would be worth mentioning fog since it has a large impact on the look of the scene. The fog I used is the easiest form of fog: linear fog. Implementing it required only adding a couple lines of code to my fragment shader. First, to get the fog, a fog factor is required. The equation for this is simply: FogFactor = (MaxFogDistance - DistanceToFragment) / (MaxFogDistance - MinFogDistance).

This fog factor is clamped between zero and one, and is then used as the alpha value in a linear interpolation between the fragment color and the fog color: Final-

Color = (1.0 - FogFactor) * FogColor + FogFactor * FragmentColor. The end result can be scene in Figure 8 and Figure 9.

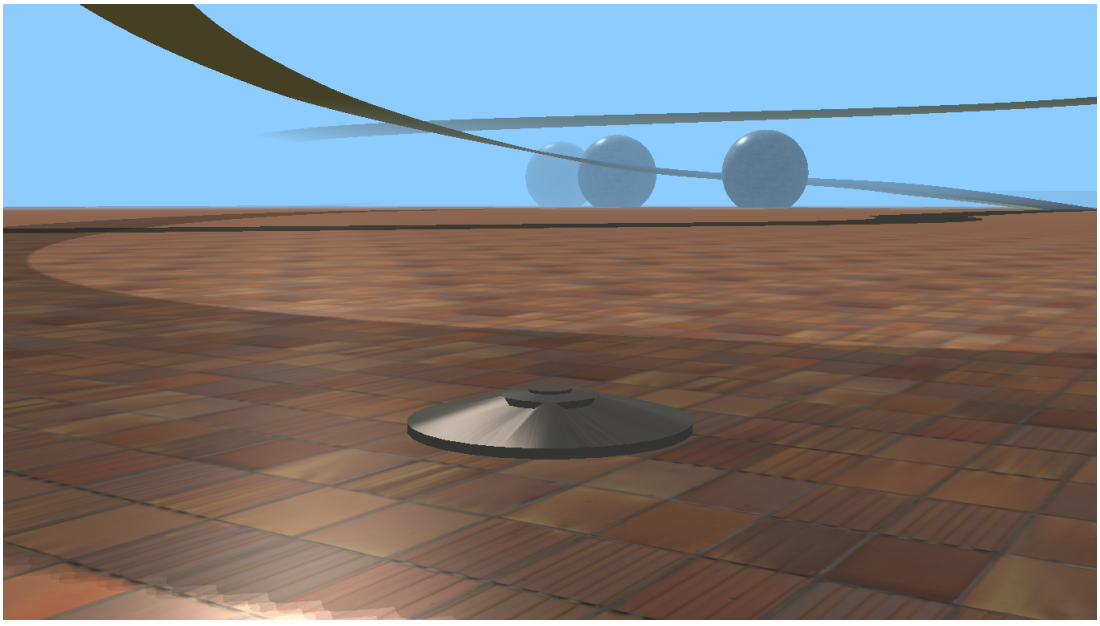

Figure 8: Track and spheres in fog.

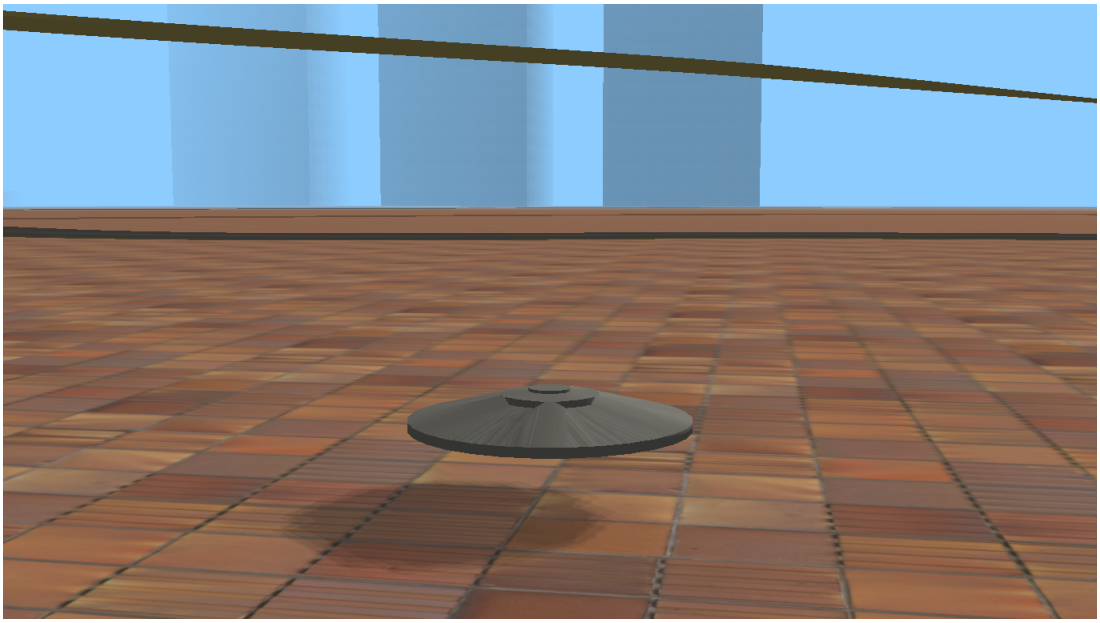

Figure 9: Buildings in fog.

4 Collision Detection

To make a racing game playable, collisions are required. Collisions govern the way a player interacts with the surrounding environment. It serves as the basis for resting motion, it allows a player to drive over terrain and roads, and it enables the player to smash into, and reflect from, walls and objects.

To handle collisions in my world, I began with one of the most simple forms of collision detection: axis-aligned bounding boxes. This offered a light-weight method to detect when I hit walls in my world. However, this method alone left much to be desired since my world contains tracks, spheres, and other odd-shaped objects. As a more permanent solution I looked into the PQP (Proximity Query Package) collisions library.

The basic idea behind PQP is to test every set of triangles between two objects for an intersection, then report the colliding triangles. In addition, it can be used to tell the distance between the closest pair of triangles between two objects. This library was exactly what I needed since it would provide a way to test against my winding roads.

In order to use PQP, I had to convert the relevant objects to be tested into `PQP_Models`. This required using the object's vertices and indices buffers with scal-

ing applied and packing them into the model with the `PQP_Real` type. These models could then be run through PQPs collision engine against each other with addition translation and rotation information (which I converted into `PQP_Real` arrays from their previous vector and matrix forms). Also, since my roads had such dense geometry, I constructed a separate track to be used in collisions that had only one tenth as many vertices. This track can be seen in Figure 10.

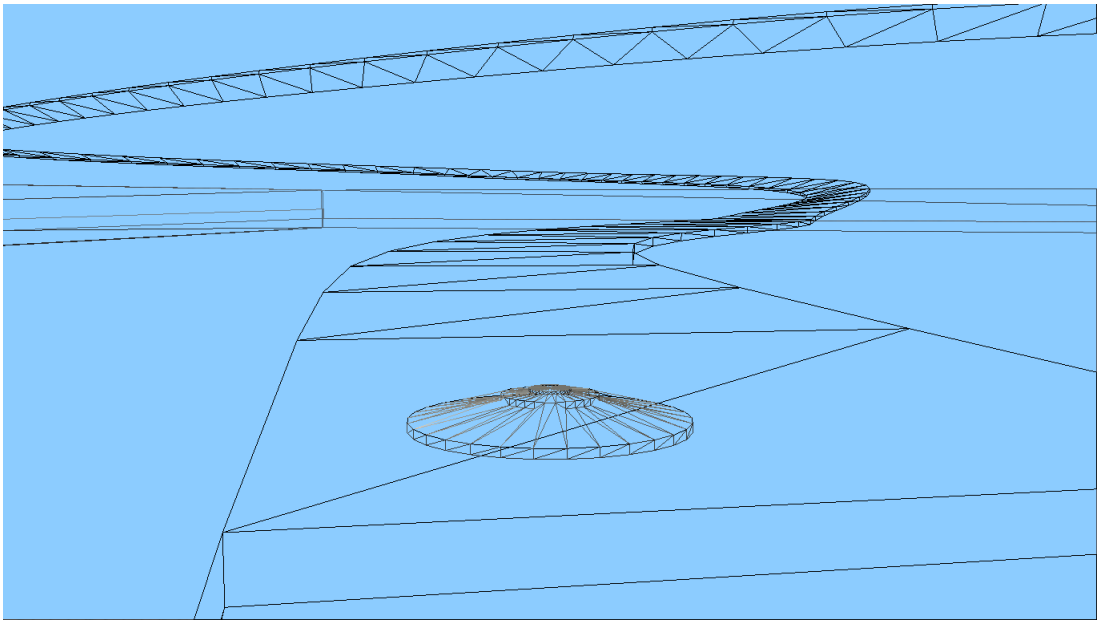

Figure 10: The reduced track mesh.

A major issue I ran into with this collision detection system was that it seemed to occasionally miss a detection on some frames. This resulted in my vehicle sometimes falling through the floor for no reason. The solution to this issue eluded me to the end and I was never able to solve it.

5 Physics

For a racing game to be complete, it needs some sort of physics to govern the way a player can move. After some reading and research, I landed on the idea of using rigid body dynamics to govern my vehicles movement.

As I stated in the introduction, I began coding my physics in the 2D case. This proved to be exceptionally easy as it only required rotations in own direction. How-

ever, when I moved into 3D, things got much trickier and proved very difficult for me, especially considering that I have no background in physics. After some struggle, though, I was able to come up with a solution that worked and seemed to govern the movement of my vehicle in a semi-believable manner.

The short version of my solution is that I represented my vehicle as a rigid rectangular prism which could have forces applied to it. I applied a forward force representing the engine force which included an acceleration force and a damping force. I then found a torque based on the forces applied to the corners of the prism and used this to determine the change in rotation of the vehicle.

I then focused on the forces that had to be applied during collisions. This would end up determining how the vehicle behaved in its resting state, how it drove over terrain, and how it reacted when hitting more vertical objects. This was the most challenging part of my physics, which I never achieved a great implementation of.

I succeeding in finding the appropriate forces to apply, however these forces proved to be too small in most circumstances since the collisions were usually detected with the vehicle part-way inside of another object, causing the vehicle to sink through objects. I attempted to fix this by using PQPs distance query and applying larger forces when the vehicle got closer to an object. This solution partially worked, resulting in a semi stable vehicle that could drive over things, however with this solution, it still has a tendency for one corner to dip into an object, resulting in the entire object following it and the rotations getting way out of control. Figure 11 shows the vehicle driving over the track how it should and Figure 12 shows the vehicle sinking into the track as described. Sadly, this is where my work had to end since I ran out of time.

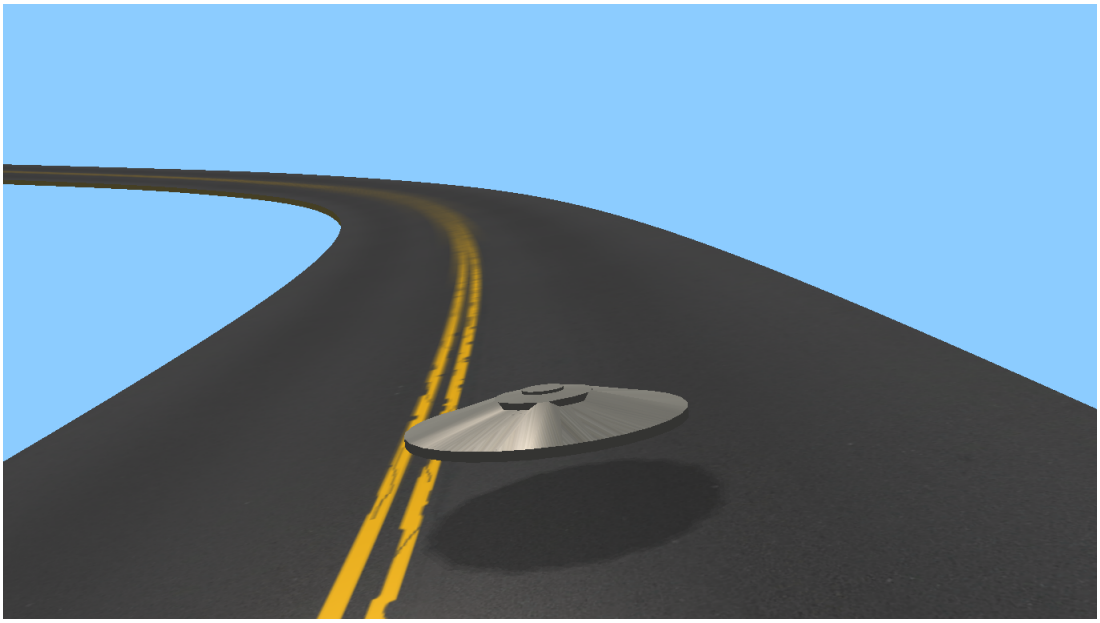

Figure 11: The vehicle driving on the track correctly.

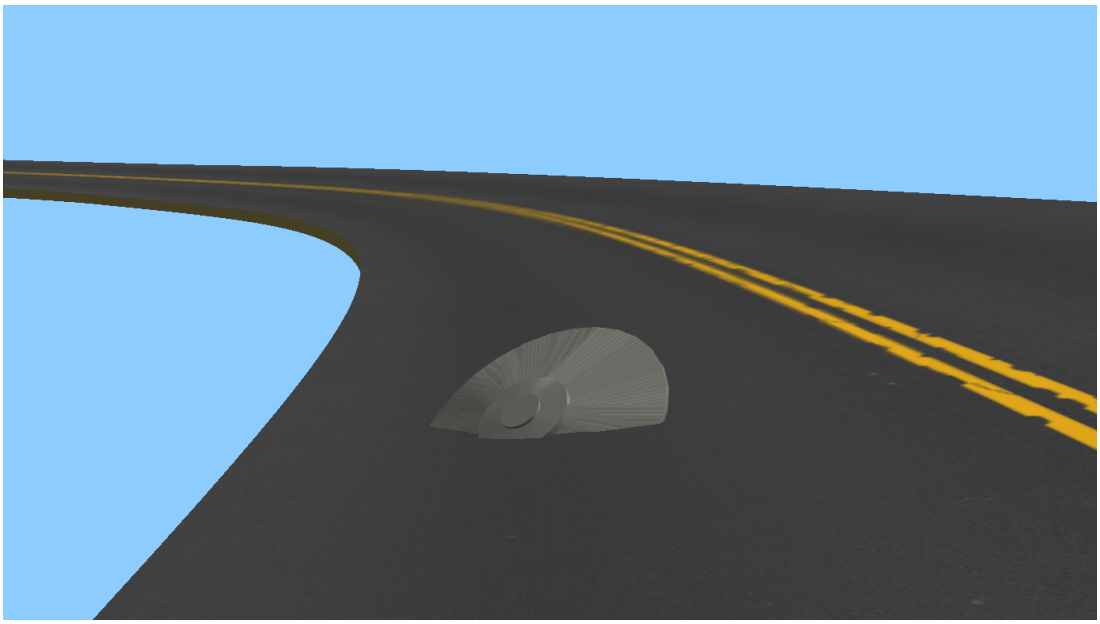

Figure 12: The vehicle falling through the track.

6 Conclusion

In the end, I did not make it as far on my project as I had hoped. However, I learned many valuable lessons, including the implementation of complex concepts and even general coding practices. I learned a fair deal about physics that I never knew in the past. I gained a substantial amount of understanding of how the pipeline in OpenGL works and of the various data structures that can be used in OpenGL. I acquired a much better understanding of how compilers work and learned how to use cmake to make compilation much quicker and more portable. I practiced object-oriented programming and gained experience in setting up a flow of data that will be more lasting. I added to my knowledge of the C++ language. Lastly, I got experience with independent programming and added to my ability to think critically.

There is so much more work that could be done on my project with more time. For example, if I had more time I would work on creating a much more rich, and full world by adding more geometry and exploring new graphics options. I would fix my broken physics and add some gameplay mechanics to make it a fun experience. I would add AI to my game or find a way to network with another player. The list could go on.

In retrospect it may have beneficial to work in a small group so that different members could have tackled the different tasks. A three-dimensional racing game is a lot of work for own person to take on, especially in light of having other classes and responsibilities. However, despite the unfinished state of things, I am still happy with what I accomplished and had a great experience doing it.

References

[1] "Fog outside." [Online]. Available: http://www.mbsoftworks.sk/index.php?page=tutorialsseries=1tutorial=15

Good place to start for implementing fog.

[2] "Physically based modeling: Principles and practice." [Online]. Available: http://www.cs.cmu.edu/ baraff/sigcourse/

Great source of information on implementing rigid body dynamics.

[3] "Tutorial 16 : Shadow mapping." [Online]. Available: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

Useful OpenGL tutorial on shadow mapping.