Modern OpenGL and Procedural Terrain Generation Exploration

Tyler Kennedy Collins
tk11br@brocku.ca
4956637

Preston Engstrom
pe12nh@brocku.ca
5228549

1 INTRODUCTION

The main purpose of this project was to explore not only the use of modern OpenGL, (version 3.3) but to experiment with the process of developing a system which could procedurally generate terrain. An additional motivation was the ability to obtain experience working with third party libraries. Some examples from among these were glm, SDL, and SOIL. Included with our project is a small C# based application which we used to explore the boundaries of Perlin Noise in relation to terrain generation. An additional two avenues that we explored were model loading, and engine building. Luckily for us, our implementation of a physics engine was simply just collision with a surface. On the other hand, our model loading system allows for us to import complex meshed objects with different materials. This project thus takes on the form of a graphics application style game, with the ability to fly a model around some world that possesses completely procedurally generated terrain.

2 LIBRARIES

Below is an explanation of the function of the main libraries included in our project, their main uses, and why we chose them over the alternatives.

2.1 GLEW

GLEW stands for The OpenGL Extension Wrangler Library. The function of GLEW is to provide some sort of open source C++ extension loading dependency. This means that it would be GLEW's responsibility to determine if certain functions are available on the hardware running a project with GLEW. If a function were not supported - it is possible GLEW would throw an error. But in a better case, GLEW would be able to translate the impossible call into something the hardware could understand and then execute it properly. The reason we chose GLEW is its constant good reviews and simplicity as well as its ability to work on multiple operation systems.

2.2 SDL

The library we chose for creating windows and handling user input was the popular SDL framework. SDL stands for Simple DirectMedia Layer, and it aims to provide access to all sorts of low level input, and output. Again, keeping with theme, SDL was chosen for its ability to work natively with C, and as such work on multiple operating systems. Another factor in choosing SDL was the availability of documentation and code examples. When learning a new library, one should not have to go scouring the depth of forums to find your input output functions.[1]

2.3 GLM

In modern OpenGL a lot of functionality has been removed for the purpose of giving you more of an ability to customize behavior. One of the things lost was the ability to simply transform any sort of view space or other that you were using. GLM does not add this back in, instead it gives you all of the tools to compute vector and matrix math for rotations, translations, and scaling. Without GLM, you would have to implement all of these methods yourself. The reason we chose GLM was its availability as well as its functionality when creating new vector and matrix objects.

2.4 SOIL

SOIL stands for Simple OpenGL Image Library which suited our needs perfectly. The same way that FreeImage functioned in class - SOIL did for us. Meaning it allowed us to quickly and easily important images for our textures and skyboxes without any issues.

3 Architecture and Class Design

Early in our design process we decided to make the switch to modern OpenGL. Knowing that this would be much more work with having to learn things like GLSL, shader language, we designed a graphics application in the form of a game. This would allow us to explore different

[1] It should perhaps be noted that the GLFW library had an Oculus Rift guide on the front page - so at least it has that going for it.

portions of modern computer graphics in many directions. This section will highlight the most important classes and their functions, why they were designed the way they were, and how they interact with modern OpenGL.

3.1 SIMULATIONGAME

This class is the main body of the entire project. From here, it was designed that any interactions from the user would be handled by SDL, and then passed to the proper external classes containing anything from character movement to camera updates. This class must also then hold reference to anything that is to be drawn on the screen. This was never a problem as we just constructed several class members with proper getter and setter methods. An interesting ability of this class was the implementation of an FPS counter. In the title of the window, the Frames Per Second count is displayed. Not only is this a very accurate count, but it will also cap the application's frames per user choice.

Since this class was the main body, it contained the game loop as well. The way we structured our game loop was fairly simple. Each loop through it would examine input, update the character, draw the world, then update the terrain, and finally handling any FPS calculations. In terms of drawing the world, everything but the model and skybox were handled as a TerrainChunk.

3.2 TERRAINCHUNK

In terms of drawing the world we thought it would be better to break up pieces of it into "chunks". This was accomplished by building an n x n lattice of points in terms of x and y, turning it into a full mesh of triangles, and assigning them some height value (Eventually accomplished by Perlin Noise). This chunk structure stored only its center position so that the outside system would be able translate the chunk via its center to the proper location in world space. Inside of this class is also where most data is sent over to the graphics card. We take advantage of an Element Buffer Object, a Vertex Array Object, and a Vertex Buffer Object to accomplish this feat. An important note is that early on we decided a large bottle neck would be constantly calling "new" or "malloc" to draw new chunks. Instead we created a way to deal with this.

3.3 TERRAINLIST

The TerrainList class is a member of the SimulationGame class and is our attempt at managing all of the chunks that we have allocated at the start of run time. The idea is to never call new on a TerrainChunk. Instead, after reaching a certain distance from a chunk, it is now marked as too far away for viewing. After this, the TerrainList class makes use of a smaller helper class, ChunkMapper, to get locations around the player in world space where chunks should be drawn. The TerrainList class then takes a chunk that is not drawing and rebases it to this given location. This effect allows the player to move smoothly through space if the open neighborhood being checked by the ChunkMapper is large enough.

3.4 Character and Camera

A camera in an OpenGL application is nothing new. The only thing that makes the design of this class relevant is that the Character class passes all of the necessary position vectors over. Another interesting note is that heavy use of GLM is needed to accomplish a fully functional camera. This is due to all of the rotation transformations that must be done about particular vectors.

3.5 GLSLProgram

Inside of the SimulationGame class there also exists a GLSLProgram object. This is a fully compiled shader class which allows for the use of uniform variables[2]. In addition to that, there is also some built in error handling inside of these objects. In reality this class mostly just compiles and binds shaders while giving a convenient way of wrapping them.

3.6 Skybox

A fun feature we chose to implement was that of a Skybox. The idea is to construct a cubemap with vector normals facing inwards, and then projecting textures onto these. The actual Skybox class is a simple one which wraps a SOIL texture loader and a particular shader for displaying the Skybox. A fun note, and interesting bug that we encountered was that you of course must turn off back face culling when drawing any Skybox at the beginning of your draw world cycle.

3.7 Memory Management

Any other classes found in the project will find explanations elsewhere in this report. It should be noted though that the purpose of most classes in the project is to avoid any sort of low level memory management as it can be tedious and quite often difficult to maintain a perfect system. Our design is completely intended to avoid this sort of trouble.

4 On Perlin Noise and Procedural Generation

5 Modern OpenGL

6 Model Loading

7 More Time

Given more time, we would have liked to explore even more concepts but some features simply did not make the cut. Below you will find a list of features that given enough time would have found their way into the application.

- The first item

[2] More information available in the OpenGL shader section.

- The second item
- The third etc ...

8 CONCLUSION