TGRA Programming Assignment 5 Project Report

Jökull Máni Reynisson

(jokull16@ru.is)

1 Introduction

1. Instructions

   Run the executable in the `Release` folder, it should work out of the box. Otherwise, the project can be compiled from the Visual Studio solution if you have the proper libraries installed (glm, glew, glfw3, assimp). They can be easily acquired using the `vcpkg` application distributed by Microsoft found on Github.

2. Control Schema

   (a) Movement
   - WASD

   (b) Camera
   - HJKL
   - X (to switch from play camera to debug camera and back)

   (c) Selection
   - Use the mouse to click objects in the scene and drag them around

3. Goal

   This project is an exercise in efficient implementation of a very simple cross platform general purpose rendering API that would be appropriate in the context possibly of making a game or an otherwise interactive simulation. The goal was to be able to render a large number of textured and possibly high poly meshes efficiently. This results in a number of important challenges that must be overcome on the path

to rendering a large number of meshes, some being a little surprising and the solutions occasionally being pretty interesting. My choice of language to implement the project in is C++ as it is familiar, efficient, is easy to set up an OpenGL development environment in with `glfw3` and has a wealth of reference material regarding implementations of larger systems. A key point is also the ability to have very granular control over how memory is utilized. I did not manage to get everything done the way I wanted it to be done and some systems had to be rewritten from scratch, resulting in a much longer and more tedious development process than I was anticipating. This resulted in the gameplay component of the project to be very underwhelming.

There is quite a bit going on, but the code should be kind of readable. I will explain some of the highlights and interesting tidbits in this report as well as touch on the architectural decisions in broad strokes.

A lot of code is based on examples from `https://learnopengl.com` which I used as a jumping off point as it is an incredible resource to get a good understanding of

2 Broad Strokes

The system is implemented such that any interactable object is an `Entity`, consisting of the usual position, rotation and scale parameters as well as a pointer to the model that is assigned to it. There is also a bounding sphere for ray casting tests that is used for the novelty of dragging entities around the screen. Entities needed to be much more performant than any other aspect as they represent the majority of visible objects on screen.

I adopted the convention from learnopengl.com to organize models as dynamic arrays of `Mesh` objects which represent the actual geometry stored in the OpenGL buffers. This allows for a very easy translation from the way that Assimp loads .obj files. Models needed a local transformation system so they have a `transform` matrix that is used to make sure that the 3D models I downloaded primarily from free3d.com would not appear strangely rotated or scaled. Assimp also can calculate the tangents and bitangents for each vertex so normal maps was an exciting prospect and this was implemented as well based on the calculations the learnopengl.com page regarding the subject.

3 Materials, Textures and Lighting

Materials used to have a lot more parameters, but I ended up narrowing them down to `diffuse`, `specular` and `shininess` as it ended up giving a much more reasonable picture if the ambient factor was controlled in the lights as a simple multiplication of the diffuse color. Textures will overtake the `diffuse` and `specular` components in the case of diffuse textures and specular textures in the shader. This did remove the possibility of recoloring the texture by using the material, but I didn't really miss it since it's simple to edit in Photoshop or Gimp and provides a less cluttered interface.

In the same vein, I was very delighted to find a nice light falloff function from an Unreal Engine paper that I credit in the fragment shader that is based only on a radius parameter and distance from the fragment. It looks pretty good and made lighting a much simpler task compared to the system I was using before which depended on a constant, linear and quadratic parameter.

The lighting model itself only implements simple directional lights and point lights using lambert and phong calculations but supports an indefinite number of them, only bounded by a `#define` in the shader. A very involved lighting display was planned but ended up being cut for time. I ended up using Phong lighting as opposed to the Blinn-Phong modification since the extra calculation of the reflected light vector wasn't really impacting performance very much and I thought that it looked nicer in some situations.

The main takeaway was that I managed to get normal maps working and looking alright. The moving lights near the start moving along bezier curves provide a nice lighting component that shows off the normal maps and the relatively high-poly Hand mesh that represents the "player". I ended up hardcoding the locations and names of each texture type that I ended up supporting in the shader since I gained a few milliseconds per frame by relying on strings as few times as possible. Since most meshes in the maze have textures, this ended up providing a lot of additional performance.

4 Key Optimizations

Pretty hilariously, most of optimization was actually minimizing the use of strings and in particular string concatenation. When setting the material parameters, which are implemented in the shaders as a struct, the original implementation relied on naive string concatenation with the name of the struct. This was taking up 36% of computation time every frame. This

unexpected inefficiency had a surprising solution where I could reduce the amount of computation required to concatenate the strings by relying on the `+=` operator instead since the compiler can make a lot more assumptions in that case. This lead me to some research that this is a known problem in video game engine programming and is often solved by having a dedicated `StringBuilder` class which provides a very efficient way to work with strings by preallocating the buffers as well as using other optimizations. The current implementation takes up around 8% of time per frame just to concatenate the strings required to set the material parameters in the shader. This could have been optimized further by never using strings at all and just relying on the handles that the shader program gives, but would've required more boilerplate code and I wanted this to be more flexible as this was not the final intended material implementation.

However, as every frame each Entity must have a correctly calculated model matrix each frame, I decided to have each entity recalculate its model matrix only when position, scale or rotation was changed. This would result in a lot of extra calculations for dynamic objects but since the majority of entities were static this reduced the number of calculations drastically. Also merely passing in the string "uModelMatrix" to the shader was using a large number of CPU cycles per frame, so I ended up caching the handle for the matrix in the shader allowing me to push through a lot more meshes per frame. I reached around 40,000 meshes rendering each frame with reasonable FPS on my main PC, which is not incredibly impressive but this number was gained through incremental performance optimizations and it was a lot of fun learning about the issues of cache misses, cache locality and the inefficiencies associated with communication to the shader.

5 Extra stuff

1. Bezier Curves

   I implemented a version of bezier curves found in `animation.h` that relies on a recursive definition that is reasonably performant, but has the entertaining feature of being type agnostic as long as the `glm` implementation of lerp, `glm::mix` can linearly interpolate it. It supports any number of control points as well. I use it to animate the two lights near the start of the maze as well as their color.

2. Picking Entities in the Scene with Ray-Casting

   The original idea was to implement an engine first and then develop

gameplay. Unsurprisingly, this was pretty over-ambitious but I don't regret a lot of the failed attempts early on as though I am not really happy with the final result, I learned a lot about graphics programming in practice. However, this is why before I even had lighting properly implemented this feature was already in the program. This took a bit of finagling as making the screen-space to camera-space to world-coordinate transformation was an almost success, but there was some slight discrepancy from the cursor position to the actual direction vector I rendered using an early version of the line rendering I implemented in the `Debug` class. This was probably due to forgetting to account for the Perspective transformation. However, after feeling reasonably confident that I understood how the transformation worked I found that `glm` had an `unProject` function that made this calculation perfectly. This was great as there is a wealth of interactive possibility using this feature. I ended up keeping it simple and having the object vary along the plane formed by the object's position and the direction vector (calculated as the vector from the camera to the cursor's world position) as the normal.

3. Fresnel Selection Effect

   This was really cool with the normal maps, but was implemented pretty early on as I wanted some visual effect to indicate selection. I researched some shader effects and the fresnel effect as implemented for usually transparent glass-type objects seemed a great fit. I ended up basing my implementation on Kyle Halladay's explanation of the effect on his blog article "The Basics of Fresnel Shading", picking some reasonable values to calculate `R` and though the linear interpolation between the fresnel color (yellow) and the Fragment color looks incorrect I thought the effect came out really nicely.

4. Skybox

   There's a lot that can be done with the skybox, but I just wanted to have some way to orient oneself easily and having a clear horizon seemed like a good idea. So the skybox is implemented as a sphere rendered without depth testing around the camera at all times with the fragment color varying based on its height, being mixed from two colors.