Implementation

I began with the shared libraries from previous assignments. I have integrated gl3w, glfw, imgui, lua, and lodepng. They are all present in the shared/ directory and compiled as external libraries with no changes. I have also integrated the framework library in the src/utils directory. I removed references to the course code to make it harder for cheaters to find the project in the future. The only file I have changed otherwise is *ObjFileDecoder.cpp*, as noted below. I also used the scene lua files from assignment 4 as a template for loading the lua file. It's changes are described under objective 1.

I think in terms of encapsulation, the renderable objects are encapsulated in their own classes fairly well. I wish I had encapsulated things like extra texture objects and drawing buffers too though. They have all ended up crowding the $Forest$ object, making it a little too large.

I have encountered issues with changing the resolutions. Specifically related to the calls to $glViewPort$ in the $Forest :: render$ method. I have to change once to the shadow map resolution, and then back to the screen resolution. I have built the program to work with a 1080p resolution, and initially launch at that, but other resolutions may not work as well. Replacing the $glViewPort(0, 0, 1920, 1080)$; call with one that uses the current resolution doesn't quite work as a fix either. On my test machine with a 1080p monitor I found that the numbers reported in the $(m_framebufferWidth, m_framebufferHeight)$ values to actually be $(1920, 1032)$, probably because of it not being fullscreen. They would then change to a 4k resolution when I dragged the monitor to my laptop's high dpi screen, but actually be the full resolution. I couldn't seem to get it to work well on every monitor size.

Loading objects / objective 1

I started loading from the lua file using A4's *scene_lua.cpp* as a template. I used the functions that set up loading from the file, and then used the actual script-bound functions as examples for when I created my own bound functions like $gr.make_list()$.

I found this took a lot of time, but worked fairly immediately when it was done. The only issue I encountered was with taking tables as an argument. I couldn't figure out how to index into a table to retrieve values, so I changed the functions to take three arguments in place of a table. It looks less clean and readable in the script, but saved development time and doesn't change functionality. This is all in a file of the same name under the src/ directory.

I had to modify the file *ObjFileDecoder.cpp* because I found it failed for many obj files that weren't quite up to spec but otherwise correct.

The actual generation of permutations of objects takes place in $MeshObject ::$ *init* inside the file *src/MeshObject.cpp*. We generate permutations by creating a separate model transform and translating it a random amount along the x and z axes. We do this to permute the objects location through the scene. We don't move through the y axis because that would make the object float or sink into the ground.

A demonstration of this is in the *geometryForest.lua* script. The buckyballs are permuted only with this method, while the other two types of objects are permuted additionally in the was described under objective 3.

I think my biggest mistake came at this point in the project. I decided to let MeshObjects take care of their own shader programs. This proved to make things like rendering them for shadows later much more difficult because I couldn't just change the shader and use the same render method.

I think the best area for the program to go next would be for it to dynamically load chunks of objects that are close to the user to create an infinite forest. This would create many smaller areas of objects and put them together, dynamically loading and unloading them as the user moves through the world. I think this would have been a much better objective than some of the other things I did, but also pretty hard. Smooth world streaming is still a big problem in open world games today.

Skybox / objective 2

This objective is pretty much just OpenGL code. It loads one sunny skybox image and samples from it using 3D coordinates and the OpenGL $GL_TEXTURE_CUBE_MAP$ functionality. We take a direction from the current camera location independent of translation so the skybox appears to never move. The code for setting up the cube map is in the *src/Skybox* object, and the shader that renders it is *Assets/Skybox*.

Variations / objective 3

We take two additional functions in the lua script that define limitations on scale and rotation.

For scale, we take the model as if it has a transform with a scale of 1 in all directions. We lock the directional permutations together to avoid wonky permutations. We generate a scale between 1 and the maximum specified scale, and then multiply that with the translation permutation that is instance specific. This can be seen in the same place as objective 1.

For rotation, we take a maximum and minimum rotation in each x, y, and z axis, and then generate a number between the two as the instance specific rotation. We add these together using a quaternion before adding them to the instance specific transform.

In practice, rotations in the x and z directions are disabled. Their code can be seen commented out inside $MeshObject :: init$. This is because they tended to create a sea of flying objects. If the original model transform didn't put the object with $(0, 0, 0)$ inside the model, then the application of x and z transforms would move it skyward. y tended to be safe, and works quite well in practice.

Inside the *geometryForest.lua* script, the stellated dodecahedron's have extra transforms applied. The large ones have extra scale transforms, and the small ones have both scale and y rotation transforms applied.

Camera Movement / objective 4

The way the camera is programmed is once the Forest object has determined the user is executing a command that will change the camera, it creates the transform and then hands it to a Camera object (from *src/Camera.cpp*). That object hides how transforms are combined from the Forest. The Forest then asks the Camera for a Projection and View transform once a frame, at which point the Camera combines them and returns them.

This allows us much more granularity in how transforms are applied, while also letting us encapsulate them away. I wish I had done the same thing for the shadow map transforms.

We can then do things like for example, have the y value of the rotation applied after the current rotation, but the x value before. This let me create a much better user movement experience.

Texture Mapping / objective 5

Phong Shading / objective 6

I translated the fragment shader from A3 to work with my program and perform the Phong shading. I modified the MeshObjects to also upload information about normal. I also had to modify the ObjFileDecoder to generate normals when an obj did not include any of it's own. It doesn't attempt to find vertex normals, just calculates face normals.

The code for this can be seen inside *src/MeshObject.cpp*, *ObjFileDecode.cpp*, and the Phong shaders inside the Assets/ folder. It should be noted that I have a friend named Phuong, so I spent the first few days of working on the project accidentally spelling things that way instead.

Shadow Mapping / objective 7

Ambient Occlusion / objective 8

Crepuscular (God) Rays / objective 9

I didn't have time for this one :(

FXAA / objective 10

The implementation for this one is pretty much entirely inside the $Assets/FXAARendererFS.glsl$ shader.

I think this algorithm was really cool to implement, and I'm pretty proud of it. The first step is deciding which pixels may have aliasing issues. We do an edge detection algorithm where we take the relative luminosities of the surrounding pixels, and then early exit if the maximum range difference is under a certain threshold ($EDGE_THRESHOLD$). If you choose FXAA RenderMode 1, you can see all of these pixels in bright red.

The next step is figuring out whether the detected edge is closer to vertical or horizontal. To do this, we take samples of relative luminosity from the 4 more cornering pixels, then calculate weighted averages and compare them. To see

this visualized choose FXAA RenderMode 2. The Vertical lines will be Purple and the Horizontal lines Yellow. In areas with angles close to 45 degrees you can see there is a combination of much of both colours.

We then take steps parallel to the line to try to sample how soon it ends. This taking steps parallel thing tripped me out at first, but it makes a lot of sense. We gather sample information along the edge and end up blurring along the edge to make the transition seem much more gradual.

RenderMode's 3 and 4 control the number of samples. 3 takes one sample, which looks good, but not that gradual along very horizontal or very vertical edges. 4 takes up to 8 samples. It stops sampling when it reaches a pixel that has a much greater difference in luminosity from the current average. It takes a weighted average as it goes, weighting further pixels much less than close ones.