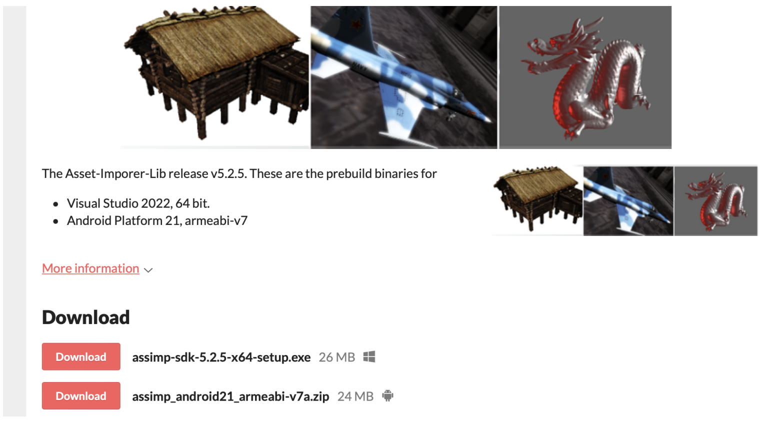
Darian Hodzic
Final Project Documentation
Independent Study: Real Time Shading

The goal of my final project was to construct an OpenGL rendering of a spaceship flying through space. My biggest focus was on recreating a starry background through the utilization of procedural shaders.

Throughout the semester I covered a total of **18 chapters** from an online textbook which can be accessed via www.learnopengl.com. Following this content, I was able to construct the rendering that I set out to accomplish.

Step 1: Importing The Model

Importing a large model of file format .obj utilizing the ASSIMP model loading library with OpenGL.


The Asset-Imporer-Lib release v5.2.5. These are the prebuild binaries for

- Visual Studio 2022, 64 bit.
- Android Platform 21, armeabi-v7

More information

Download

Download assimp-sdk-5.2.5-x64-setup.exe 26 MB

Download assimp_android21_armeabi-v7a.zip 24 MB

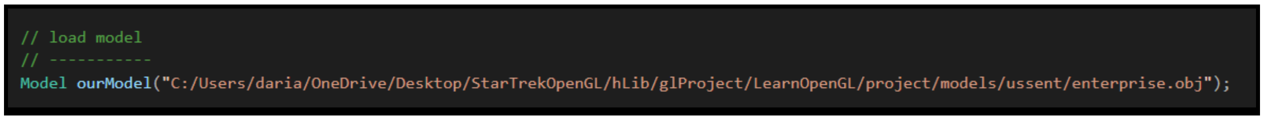
https://kimkulling.itch.io/the-asset-importer-lib

I found the spaceship model that I wanted from the following link:
https://sketchfab.com/3d-models/uss-enterprise-d-star-trek-tng-e3118c97914342b3ad7dd957c4b4ce4e

Step 2: Loading The Model In An Environment Lit With Phong

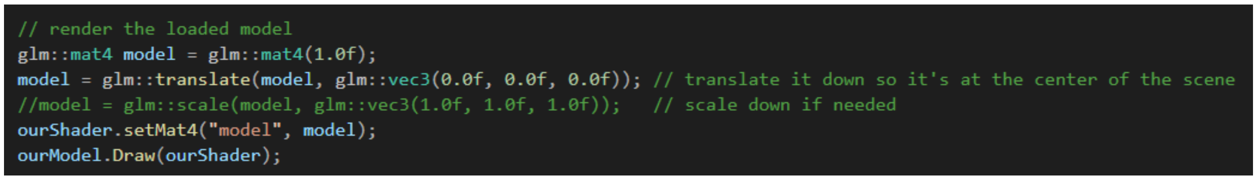
The load time of my model is approximately 1.30 minutes on average. The following code provides implementation details on how I accomplished the loading of the external .obj file and implemented the lighting component.

By utilizing the Model class provided by the online textbook, I was able to create a model object and load directly from the original .obj file that was downloaded.



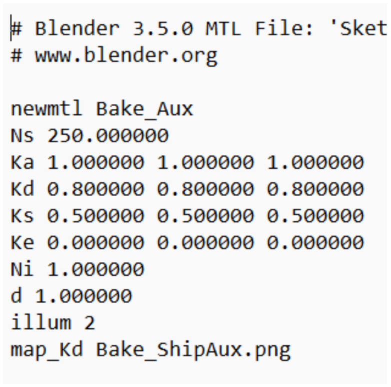
```
// load model
// -----------
Model ourModel("C:/Users/daria/OneDrive/Desktop/StarTrekOpenGL/hLib/glProject/LearnOpenGL/project/models/ussent/enterprise.obj");
```

Rendering the model was straightforward where I used the same process presented in the online textbook.



```
// render the loaded model
glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f)); // translate it down so it's at the center of the scene
//model = glm::scale(model, glm::vec3(1.0f, 1.0f, 1.0f));   // scale down if needed
ourShader.setMat4("model", model);
ourModel.Draw(ourShader);
```

I noticed that the textures were not being applied properly to the spaceship, so I found a workaround through modifying the originally downloaded .mtl file which is associated with the model. In this file, I made sure to map the Kd values of each material to its corresponding texture file which is of .png format. For example, in the .mtl file we can see a newmtl (new material) defined as Bake_Aux. I simply utilized map_Kd <texutre_file_name> to assign the correct texture.



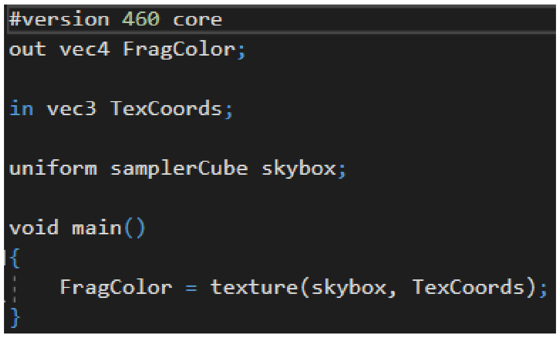
```
# Blender 3.5.0 MTL File: 'Sket
# www.blender.org

newmtl Bake_Aux
Ns 250.000000
Ka 1.000000 1.000000 1.000000
Kd 0.800000 0.800000 0.800000
Ks 0.500000 0.500000 0.500000
Ke 0.000000 0.000000 0.000000
Ni 1.000000
d 1.000000
illum 2
map_Kd Bake_ShipAux.png
```

Step 3: Creating A Skybox

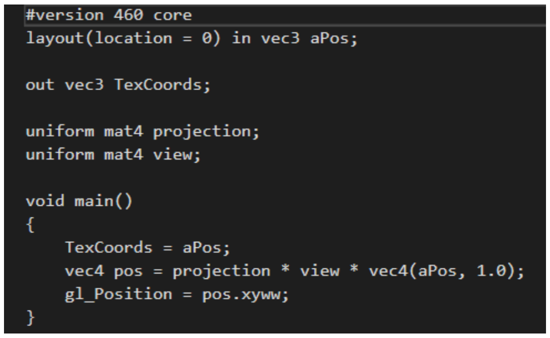
I downloaded a set of .png files which could be applied to my cubemap implementation. *https://opengameart.org/content/space-skyboxes-0*. Each face of the cubemap would be assigned one of the associated .png files from this download.

The following code provides the implementation details on how the cubemap was created and how each .png file was assigned to the corresponding faces.

Firstly, I created a skybox fragment shader (left) and vertex shader (right).



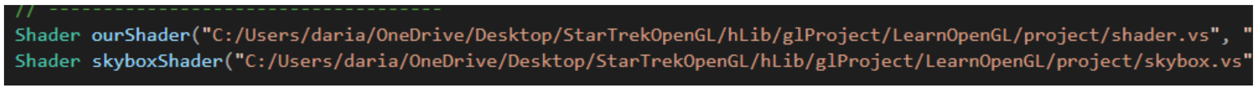
```
#version 460 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

void main()
{
    FragColor = texture(skybox, TexCoords);
}
```



```
#version 460 core
layout(location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aPos;
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;
}
```

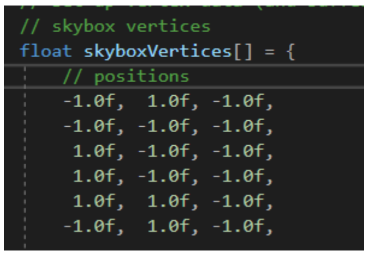
The vertex shader is fairly trivial as we assign our texture coordinates to the corresponding vertex positions on the later defined cubemap.

Within the main ogl.cpp file, I built and compiled the main lighting shaders and skybox shaders through usage of the textbook's Shader class.



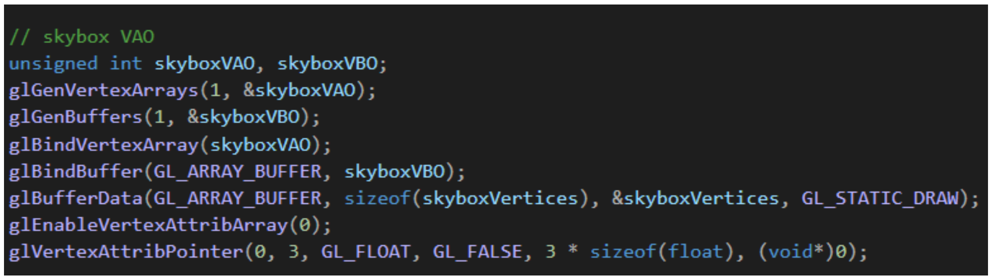
```
// ------------------------------------
Shader ourShader("C:/Users/daria/OneDrive/Desktop/StarTrekOpenGL/hLib/glProject/LearnOpenGL/project/shader.vs", "
Shader skyboxShader("C:/Users/daria/OneDrive/Desktop/StarTrekOpenGL/hLib/glProject/LearnOpenGL/project/skybox.vs"
```

For the skybox itself, the vertex positions of our cubemap are defined in an array.



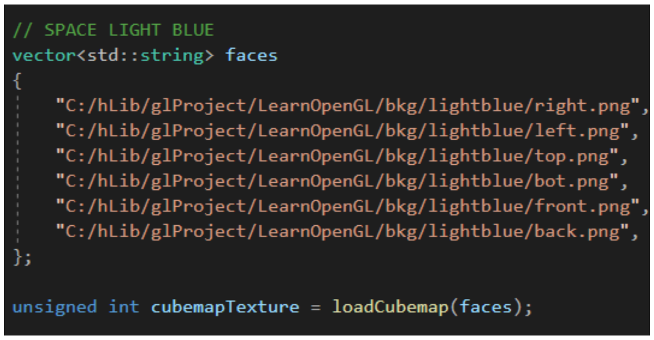
```
// skybox vertices
float skyboxVertices[] = {
    // positions
    -1.0f,  1.0f, -1.0f,
    -1.0f, -1.0f, -1.0f,
     1.0f, -1.0f, -1.0f,
     1.0f, -1.0f, -1.0f,
     1.0f,  1.0f, -1.0f,
    -1.0f,  1.0f, -1.0f,
```

The standard procedure of initializing a vertex array object and vertex buffer object for our skybox follows.



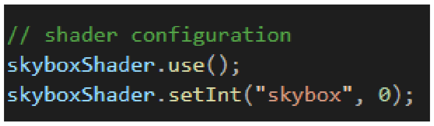
```
// skybox VAO
unsigned int skyboxVAO, skyboxVBO;
glGenVertexArrays(1, &skyboxVAO);
glGenBuffers(1, &skyboxVBO);
glBindVertexArray(skyboxVAO);
glBindBuffer(GL_ARRAY_BUFFER, skyboxVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(skyboxVertices), &skyboxVertices, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
```

The final steps before rendering our skybox involved loading the skybox textures themselves. We create a vector of string objects corresponding to the paths of each face of the cubemap. Our cubemapTexture is initialized as the result of passing the vector of string objects into a function called `loadCubemap`. This function was sourced from the textbook directly in the cubemap chapter.



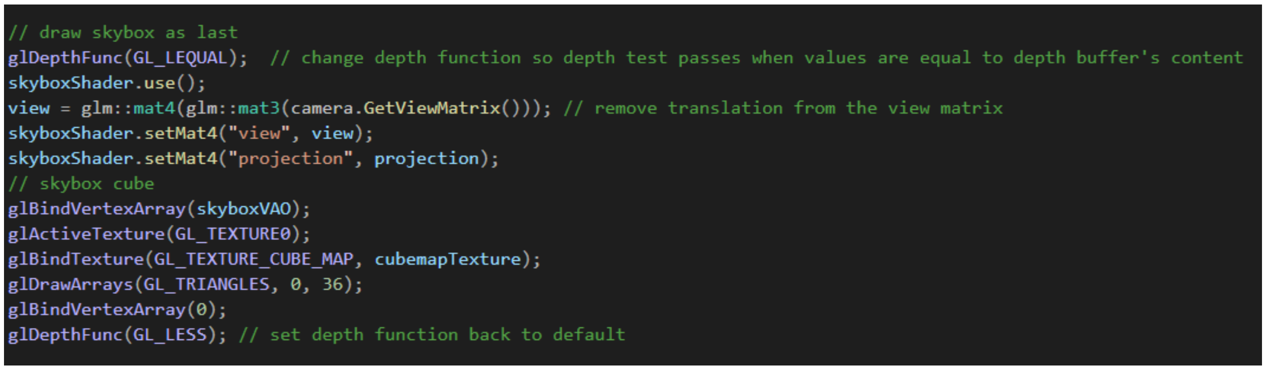
```
// SPACE LIGHT BLUE
vector<std::string> faces
{
    "C:/hLib/glProject/LearnOpenGL/bkg/lightblue/right.png",
    "C:/hLib/glProject/LearnOpenGL/bkg/lightblue/left.png",
    "C:/hLib/glProject/LearnOpenGL/bkg/lightblue/top.png",
    "C:/hLib/glProject/LearnOpenGL/bkg/lightblue/bot.png",
    "C:/hLib/glProject/LearnOpenGL/bkg/lightblue/front.png",
    "C:/hLib/glProject/LearnOpenGL/bkg/lightblue/back.png",
};

unsigned int cubemapTexture = loadCubemap(faces);
```

We make sure to configure our skybox shader by setting the texture unit of our uniform samplerCube variable `skybox` to zero. This uniform is defined in the skybox fragment shader.



```
// shader configuration
skyboxShader.use();
skyboxShader.setInt("skybox", 0);
```

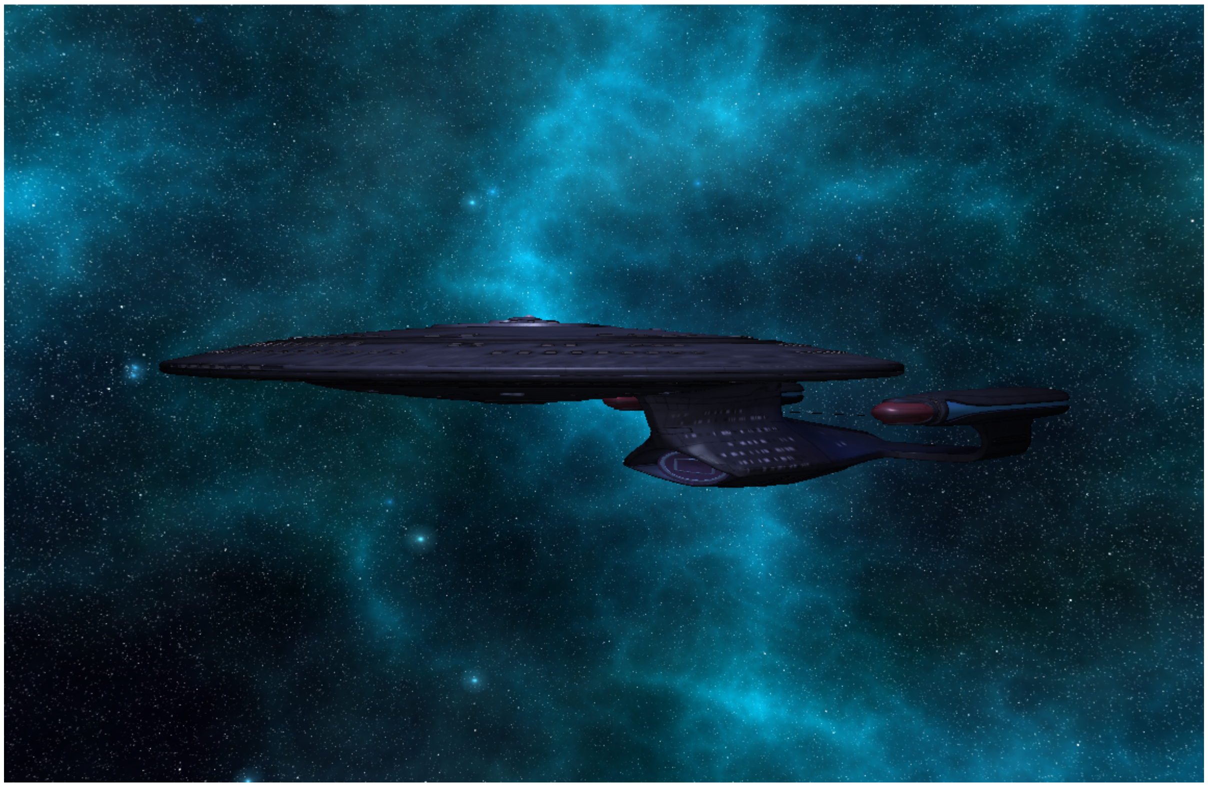
The final step was to render the skybox itself in the render loop alongside our model.



```
// draw skybox as last
glDepthFunc(GL_LEQUAL);  // change depth function so depth test passes when values are equal to depth buffer's content
skyboxShader.use();
view = glm::mat4(glm::mat3(camera.GetViewMatrix())); // remove translation from the view matrix
skyboxShader.setMat4("view", view);
skyboxShader.setMat4("projection", projection);
// skybox cube
glBindVertexArray(skyboxVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
glDepthFunc(GL_LESS); // set depth function back to default
```

We use our skyboxShader and set the corresponding view and projection matrices where our view matrix is simply the camera's view matrix. The following steps are standard procedure for working with textures as we bind our skybox vertex array object that was defined earlier and make sure to set GL_TEXTURE0 as our active texture since we are utilizing texture unit zero. Finally, the cubemap texture which we previously loaded is bound and we draw the skybox.

Result:



The final implementation can be found at *https://github.com/dhodzic1/StarTrekOpenGL*