Coursework 2 Report

Visual Scene:

For the scene, we decided to render a rocket launchpad with a rocket firing off on command in the night.

Complex Object:

The complex object comprises of two desktop monitors mounted with the help of 5 capped cylinders and two cubes. The first step was to initialize these objects with the help of make_cylinder() , make_cube() methods respectively. This follows the approach of complex model building wherein several primitives are used to build a singular complex object. The shape, angles and positioning of the components require a slight deviation from the base primitives; hence matrix translation is used with scaling. Following this, a series of concatenation calls are made to initially concatenate the base(stand) and finally the monitors into one complex object. Figure 1 shows the complex object.

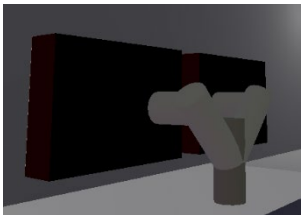

Figure 1 – Complex Object

Perspective Projection:

To allow the window to resize and for all the projections to remain intact we take the framebuffer and retrieve its current size on every frame, we the use the variables fbwidth and fbheight, and send the values of the perspective to a function in mat44 which will calculate the perspective projection matrix. This projection value will then be sent alongside the camera matrix and model matrix to the vertex shader to calculate the new position of any given vertex. Figures 2 and 3 show the program run in 2 different sizes to demonstrate the perspective projection and resize ability:

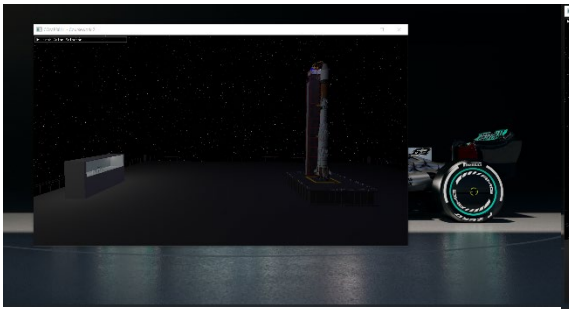

Figure 2 – Original Window

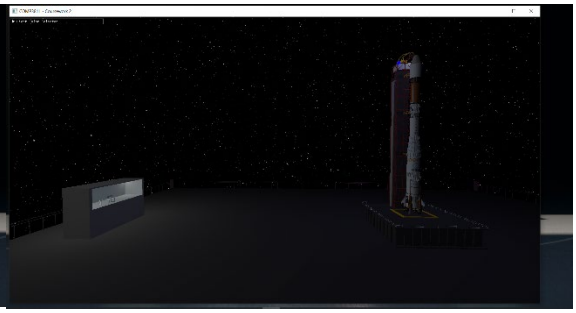

Figure 3 – Window resized

3D Camera:

To develop the camera, we began by initializing the camera controls struct and added a few additional parameters such as the x and y position and a modifier variable. We then modified the user input function and added a few more statements to cover the new keyboard controls. Once we had these, we used the tutorial posted by Brian Will (Will, 2019) in conjunction with the tutorial posted on LearnOpenGL (Vries, 2020) to develop the code needed to calculate the camera translation matrixes. The approach we followed was to calculate a new x, y, z value and then use the vector of these 3 values to form a camera transform matrix. To demonstrate we have the code used for the zoom translation shown below:

```
if (state.camControl.actionZoomIn) {
        state.camControl.y += sin(state.camControl.theta) * kMovementPerSecond_ * dt * state.camControl.mod;
        state.camControl.x -= cos(state.camControl.theta) * sin(state.camControl.phi) * kMovementPerSecond_ * dt *
state.camControl.mod;
        state.camControl.radius -= cos(state.camControl.theta) * cos(state.camControl.phi) * kMovementPerSecond_ *
dt * state.camControl.mod;
}
```

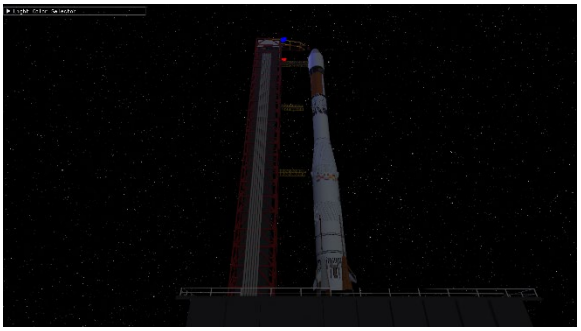

Figure 4 – Camera View

We however altered the code a little to allow for the camera zoom and movement to be relative to the players viewing direction. Therefore pressing 'w' to go forward will make the y increase if the player is looking up. We did this to allow the user to navigate around the scene more easily in an intuitive 'free cam' manner. Figure 4 shows the view from the camera initially and if you observe the other figures in this document, you will see different camera angles and position, all demonstrating the 3d camera we have developed.

Vectors and Matrices:

Perspective projection, translation, scaling all make use of matrices and vectors and manipulate fundamental vector and matrix operations such as addition and multiplication. These fundamentals are implemented in the classes mat22, mat44, mat33, vec2, vec3 and vec4. To construct the matrixes for rotation we used the source provided on BrainVoyager (BrainVoyager, 2020) and for the perspective projection matrix we used (Scratchapixel, 2022).

Shading:

To implement shading the Blinn model was used which comprises of three primary components: ambient, diffuse, and specular wherein ambient refers to the presence of a perpetual light - which in this case was moonlight, diffuse is the impact of a light on the surrounding surfaces and specular is a bright spot at which it is concentrated. To implement these components variables corresponding to ambient, diffuse, and specular strength were introduced in the shader. Further, normal vectors (vectors perpendicular to the vertex's surface) were initialized and passed to the shader. The dot product of the normalized (unit length) normal with vectors corresponding to the direction of light is used to calculate the impact of light on each fragment [1]. Figure 5 shows one such light source and the shading around it. To develop this, we used the LearnOpenGL tutorial on basic lighting (Vries,2020)

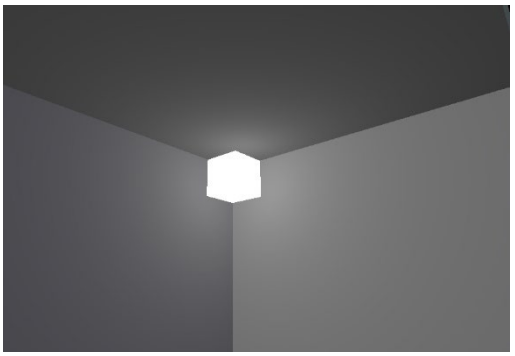

Figure 5 – Phong Lighting

Animated Object:

For our animation we translate the rocket by a calculated amount on each iteration. This amount increases exponentially but has been clamped at certain points to give a more realistic launch effect. We also did not want the rocket launching too fast to allow the scene to be appreciated in its full capacity when the launch is occurring. Especially noticing the lights at the top of the launchpad shining on the rocket as it launches past; gives a very nice effect to the viewer. The animations can be controlled using the arrow keys as well as the number keys to allow for versatile control independent on whether the machine it is ran on has a full-size keyboard or not. The arrow key controls are as follows:

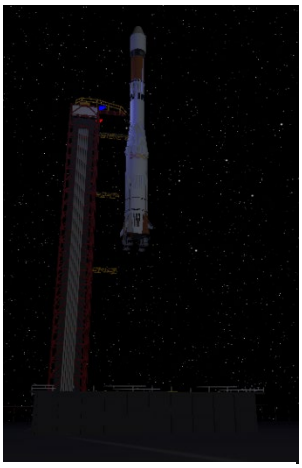

Figure 6 – Animated Rocket Mid Launch

- 2 / Up Arrow – Play/Pause
- 1 / Left Arrow – Slow down animation
- 4 / Right Arrow – Speed up animation
- 3 / Down Arrow – Reset animation speed

The animations work by passing a translation matrix to the model2world this translation matrix uses a y value of rktHeight which is calculated using the following code:

```
float x = 1.1f;
if (state.animControl.animation) {
        if (rktHeight < 10) {
                x += 0.01f;
        }
        rktHeight += (rktLast / x) * (0.015f * state.animControl.mod);
        rktLast = rktHeight;
}
```

Here the value of *state.animControl.mod* allows us to modify the speed of the animation based on the user controls. In addition, we can see how below the height of 10 units the speed increases at a slower yet exponential rate which then increases past that height. This design choice was made though trial and error and after having observed many launches to pick the most appealing option. Figure 6 shows the translated rocket in the middle of its launch sequence.

Texture Mapping:

The provided texture is mapped on the monitors of the complex object. This is achieved by creating another flat cube, that sits on the face of the monitor, which is textured. Texture coordinates of values (0,0), (0,1), (1,0), (1,1) are mapped. In cube.hpp, create_cube_tex() method was implemented to essentially draw a flat cube with these texture coordinates corresponding to every vertex of the triangles that form a cube face. To load the actual texture, load_texture_2d() in mesh.hpp uses the stb_image library's load function. Error handling is addressed by displaying an error message in case the load fails unexpectedly. The shaders and buffer arrays were altered to accommodate the texture coordinates (with the help of sampler2D in the fragment shader). Finally, in the main loop, active texture is set to TEXTURE0, the loaded texture is bound, and the vertex array is drawn. For the external rocket object, texture coordinates were already provided, so they were simply loaded along with vertex coordinates and the same procedure was followed from thereon. Figure 7 shows the screen textured using the provided coursework image.

Figure 7 – Provided texture on the screen

Blinn-Phong Lighting:

To implement Blinn-Phong lighting we expand on the Phong lighting model shown above. Bling-Phong takes a different approach to the calculation of the specular model. Here instead of using the reflection vector we use a halfway vector that is "a unit vector exactly halfway between the view direction and light direction" (Vries, 2020). We use the alignment of the halfway vector and a surfaces normal vector to calculate the specular contribution. We use the following code to calculate the new specular component:

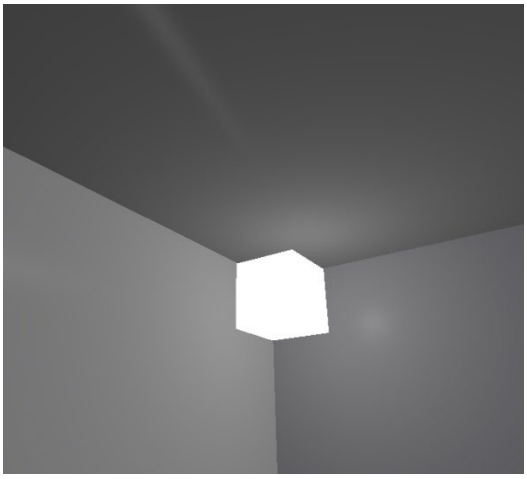

Figure 8 – Bling-Phong Light

```
vec3 halfDir = normalize(lightDir + viewDir);
float spec = pow(max(dot(normal, halfDir), 0.0), uShininess);
vec3 specular = light.specular * (uSpecular * spec);
```

We now have a stronger reflection from the surface due to this new specular component as shown in figure 8. In the scene we have chosen to keep one of the interior lights without a specular component and one with to demonstrate both types of lighting models. The lights at the top of the launch pad use bling-phong and the new specular can be seen as reflected on the rocket and the structure. The cube shown in figure 8 also has a majority emissive material and therefore you can see it gives the impression of a light in the room. This also holds true for the 2 lights on top of the landing pad structure.

Figure 9 – Table with Diffuse material

As shown in figure 9, the table in the room is of a mainly diffuse material and therefore lacks the shine which the room presents. The room itself and the ground on which the

platform is placed has a mainly specular material and therefore has strong specular components from the lights. This is also demonstrated in figure 8.

Multiple Light Sources:

In the scene we have 5 light sources all point lights. 2 have been discussed above as interior lights and can be seen faintly in figure 10, In addition to this we have 2 'spotlights' on the top of the launchpad which light up the rocket which can also be seen in figure 10. Finally, there is a point light up high on the right side of the scene which has a very low brightness and serves to provide a faint ambient light to the scene and simulate moonlight during the night. These lights all use the bling-phong lighting model as described above.

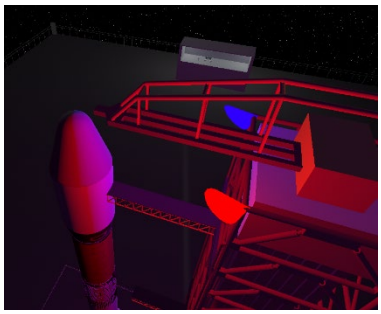

Figure 10 – Multiple point lights

External Object:

In loadobj.cpp, the load_wavefront_object() method is used to load the two external objects, namely Scene.obj and Rocket.obj. The .obj files contain data for vertices, normals and texture coordinates. This data is parsed into arrays and returned as a SimpleMeshData object to main. This object is scaled using make_scaling() and translated to place it into the scene. The imported objects can be seen in figure 11.

Figure 11 – Imported Objects

Transparent Object:

A glass window is drawn in the viewing box wherein GL_BLEND is used to achieve a "transparent" look. Firstly, a cube with the dimensions of the viewing box is created followed by a vao for the same. In the main loop, GL_BLEND is enabled prior to the call to draw arrays and glBlendFunc() is used to set the source and destination factors where source corresponds to the glass and destination to the scene behind it. This yields the following transparent window shown in figure 12.

Figure 12 – Transparent Window

ImGUI:

To develop the ImGUI interface for the program we began by importing the required library and configuring the premake files to allow us to use the newly added module. We decided to allow the user configurability in the lighting of the scene to allow them to increase or decrease the brightness of select lights as they wish. We then added further features to allow specific control over the lights and their respective colors. This provided the user with a simple yet effective interface that they could use to control the look of the scene. We chose a dark theme with the program to match the ambience of the scene we have developed.

The code begins by initializing the ImGUI window and the parameters it will edit, these include Boolean vectors for the toggles which select the lights as well as vec4 arrays for the color picker.During execution these values are retrieved from the interface and passed to the fragment shader as required, to change the color of the emissive component of the lights. We also pass the color data to the lightColor variable which defines the color of the point light. This then allows full control over the color of the scene. The code below shows how we display the ImGui interface:

```
//imgui
// ImGUI window creation
ImGui::Begin("Light Color Selector");
// Text that appears in the window
ImGui::Text("Tick the box to change lights!");
ImGui::Checkbox("Interior  Lights", &temp);
ImGui::SliderFloat("Brightness", &lightBrightness[0], 0.1f, 5.0f);
```

```
ImGui::ColorEdit4("Color", color);
ImGui::Checkbox("Launchpad 1", &temp1);
ImGui::SliderFloat("Brightness 1", &lightBrightness[1], 0.1f, 5.0f);
ImGui::ColorEdit4("Color 1", color1);
ImGui::Checkbox("Launchpad 2", &temp2);
ImGui::SliderFloat("Brightness 2", &lightBrightness[2], 0.1f, 5.0f);
ImGui::ColorEdit4("Color 2", color2);
// Ends the window
ImGui::End();
```

The ImGUI interface can be seen in figure 13. In this figure you may also see how they light colors differ from the ones shown in other screenshots.

We used the video tutorial developed by Victor Gordan (Gordan, 2021) as a reference and guide when developing this part of the program.

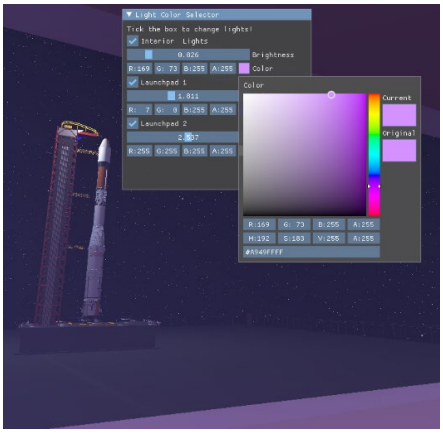

Figure 13 – ImGUI interface

Hierarchical Modelling:

The hierarchical model we developed was a fan which has a spinning blade but also sits on a pedestal which rises and falls continuously. The fan blades and motor were separate objects from the base and then respectively had transformations applied to them. We begin by transforming the motor up and down using the following code which takes the sin of the angle calculated previously using dt (delta time) and use this value along side its default translation to animate the motor. The code for motor movement is listed here:

Figure 14-

Figure 15-

Hierarchical Model

```
Mat44f motorTransform = make_translation({ 5.1f, 0.82f + (sin(angle)/16), 21.6f});
```

We take the transformation matrix of the motor and apply that to the blade object too. This allows the blades to 'follow' the motor as it rises and falls as shown in figures 14 and 15. If you observe the 2 figures you will notice how there is an arm extended from the base as the motor rises and the blades have changed rotation too. To rotate the blades, we simply used the following rotation matrix to calculate the transformation:

```
model2world = motorTransform * make_translation({ 0.f, 0.105f, 0.f }) * make_rotation_x(angle * 20.f);
```

Screenshots:

To program the screenshot function, we used the git repository provided by vallentin (vallentin, 2018) as a reference to see how the screenshot code worked and then adapted this code and rewrote it for our own purposes. Principally it is similar and follows the same steps to capture the frame. We begin by capturing the key press as we have done before and use the F12 key input to call the function saveScreenshot() which is defined in screenshot.cpp. This function beings by retrieving the viewport and initializing the width and heigh based off the size of the frame. We then use the following code to capture the pixels in the viewport:

```
char* data = (char*)malloc((size_t)(width * height * 3));
glReadPixels(0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE, data);
```

Once we have the pixel data stored, we use stbi_write_png to write the data to a file. The file name is determined using the current date and time as defined in the getScreenshotName() function. We use the following code to construct this name:

```
strftime(basename, 30, "%Y%m%d_%H%M%S.png", localtime(&t));
```

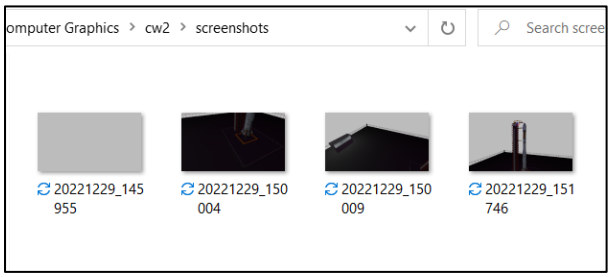

Figure 16 – Screenshots Folder

With the name we add a suffix of "screenshots/" to place all the screenshots taken neatly in a folder as shown in figure 16

Skybox:

Lastly, for environment textures, a cubemap - with a texture mapped onto each face - is used. To implement such a cubemap, new fragment and vertex shaders were initialized that make use of the samplerCube variable instead of sampler2D and a regular cube was created. Textures corresponding to six faces of the cube are loaded and bound as one GL_TEXTURE_CUBE_MAP in the main loop to give the impression that the scene is within the textured cube. The new shaders are constructed so that the skybox does not move but is scaled based on the persepective projection relative to the camera. The skybox can be seen in Figure 17 and the use of multiple shaders can be see in Figure 18.

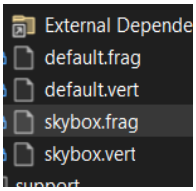

Figure 18 – Multiple shaders

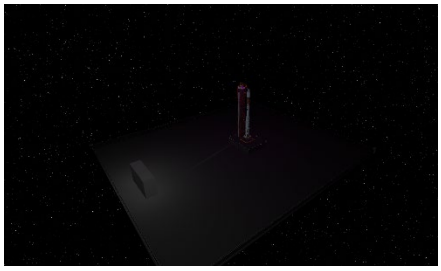

Figure 17 - Skybox

Appendix

External 3D Models:

https://www.cgtrader.com/free-3d-models/space/spaceship/rocket-launch-pad - launchpad

https://free3d.com/3d-model/rocket-v1--141304.html - rocket

https://www.turbosquid.com/3d-models/black-table-model-1818872# - table

https://www.cgtrader.com/free-3d-models/exterior/other/wire-fence-ef09c94d-0852-481e-9651-93e1e12abbcf - fence

References:

1. BrainVoyager 2020. Spatial Transformation Matrices. Brainvoyager.com. [Online]. [Accessed 22 November 2022]. Available from: https://www.brainvoyager.com/bv/doc/UsersGuide/CoordsAndTransforms/SpatialTransformationMatrices.html#:~:text=The%204%20by%204%20transformation,in%20the%20first%20three%20columns..
2. Scratchapixel 2022. The Perspective and Orthographic Projection Matrix. Scratchapixel.com. [Online]. [Accessed 22 November 2023]. Available from: https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/building-basic-perspective-projection-matrix.html.
3. Vries, J. 2020. LearnOpenGL - Basic Lighting. Learnopengl.com. [Online]. [Accessed 29 November 2022]. Available from: https://learnopengl.com/Lighting/Basic-Lighting.
4. Vries, J. 2020. LearnOpenGL - Camera. Learnopengl.com. [Online]. [Accessed 25 November 2023]. Available from: https://learnopengl.com/Getting-started/Camera.
5. Will, B. 2019. OpenGL - camera movement. YouTube. [Online]. [Accessed 25 November 2022]. Available from: https://www.youtube.com/watch?v=AWM4CUfffos&ab_channel=BrianWill.
6. Vries, J. 2020. LearnOpenGL - Advanced Lighting. Learnopengl.com. [Online]. [Accessed 18 December 2022]. Available from: https://learnopengl.com/Advanced-Lighting/Advanced-Lighting.
7. vallentin 2018. GLCollection/screenshot.cpp at master · vallentin/GLCollection. GitHub. [Online]. [Accessed 25 December 2022]. Available from: https://github.com/vallentin/GLCollection/blob/master/screenshot.cpp.
8. Gordan, V. 2021. ImGui + GLFW Tutorial - Install & Basics. YouTube. [Online]. [Accessed 28 December 2022]. Available from: https://www.youtube.com/watch?v=VRwhNKoxUtk&ab_channel=VictorGordan.
9. Vries, J. 2020. LearnOpenGL - Cubemaps. Learnopengl.com. [Online]. [Accessed 20 December 2022]. Available from: https://learnopengl.com/Advanced-OpenGL/Cubemaps.

Table of Contributions and Tasks

Task	Completion	% Safwan – ed18src	% Yash – sc19yvs
Complex object constructed in code	Yes	50%	50%
Perspective projection	Yes	100%	0%
First-Person 3D style camera	Yes	100%	0%
Vector / Matrix Implementation	Yes	70%	30%
Basic Lighting – Phong	Yes	100%	0%
Advanced Lighting – Bling-Phong	Yes	100%	0%
Animated Object	Yes	100%	0%
Animation Controls	Yes	100%	0%
Object with mainly diffuse	Yes	100%	0%
Object with mainly specular	Yes	100%	0%
Object with mainly emissive	Yes	100%	0%
Texture loading	Yes	0%	100%
Object texturing	Yes	10%	90%
Multiple Light Sources	Yes	100%	0%
Object loading (.obj)	Yes	100%	0%
Transparent Object	Yes	100%	0%
Hierarchical Modelling	Yes	100%	0%
Complex Animation	No	0%	0%
Multi-Texturing	Partial	0%	100%
ImGUI	Yes	100%	0%
Screenshots	Yes	100%	0%
Custom Shading Model	No	0%	0%
Multiple Shaders	Yes	80%	20%
Cubemap (Skybox)	Yes	80%	20%