Graphics CW2 Report

Michael Wright – 201320390
Luke McDowell – 201319894

December 2022

1 Introduction

In this report, we will cover how we have met the appropriate criterion for each band and showcase the features we have implemented in our skatepark scene containing a bowl, ramps, rails and skateboards.

2 Band 40% - 50%

2.1 Vectors and Matrices

To begin implementing the camera object we first had to make the appropriate vector and matrix functions and operators. We have used our solutions to previous exercises for the `Mat44f` times operator, translation matrix and other operators from these exercises. Translation and rotation matrices were also implemented. The newly added functions include a scaling function, a `Vec3f` cross product function and a `Vec3f` normalize function. For the `cross_product` function we have implemented these equations:

$$\begin{bmatrix} A_yB_z - A_zB_y \\ A_zB_x - A_xB_z \\ A_xB_y - A_yB_x \end{bmatrix}$$

Where A and B are the 2 `Vec3f` inputs, and the 3 x 1 matrix is the returned `Vec3f`. For the normalize function we have implemented the following code:

```
return {(input.x / len), (input.y / len), (input.z / len)};
```

Where input is the input `Vec3f` and `len` is the length of the input vector which is obtained from the length function.

2.2 Camera

To implement our camera we have created a class that contains Boolean values that correspond to each direction of movement as well as speed up and slow down. These values are updated using key callbacks through a method within the class which is called in a callback function. Each frame, the camera object will check the values of these booleans and update its position accordingly, the speed of which is controlled by the `cameraSpeed` attribute which is frame rate independent and defined in the camera class.

The users' mouse movement is captured through another callback function, it uses the x and y positions of the users' mouse within the window to compute values for pitch and yaw. Then using some simple trigonometry, a direction vector can be computed which after it has been normalized can be used as the camera's front vector. This corresponds to the direction the camera is pointing.

Using the camera position, camera front and camera up (always points up in the scene) we can compute a new matrix which we will use for our view matrix. This will allow us to use user input to move around the scene, as we will recompute this matrix every frame. Here is how that matrix is calculated.

$$\begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

R, U, D and P are all `Vec3f` values. P is the camera position, D is the camera direction vector, U is the relative camera up vector and R is the relative camera right vector.

2.3 Complex Objects

We have included a few complex objects in our scene, all of which use the `SimpleMeshData` struct as a means to construct them. Firstly we built a function which can construct one eighth of a sphere, it does this by taking a triangle and then recursively generating smaller triangles within the original. Then using a normalising function you can normalize every point on this triangle relative to a specific point and to a specific length which would be considered the radius in this example as we are constructing a sphere.

Figure 1 shows an inward facing part of a sphere, but we can just as easily generate an outward facing one by altering the order of the initial input triangle points from `CW` to `CCW` or vice versa. The amount of triangles generated can be altered by adjusting the recursion depth of our function, we have used a depth of 7 as we feel this gives us a good balance between the smoothness of the sphere and performance.

We have used this function to generate spheres for our lamps which light the scene as well as for our main complex object, which is a skate park bowl. This is a concatenation of a few different `SimpleMeshData` structs, which include the sphere generation, a half cylinder, a join from the edge of the sphere to a constant point and 5 simple rectangles to enclose it within a box.

Figure 2 is our bowl as a triangle mesh render so you can more easily see the individual parts that make up the whole object. As mentioned we have included other complex objects in our scene such as full spheres, rails and ramps, you can see these in our video. This complex object as well as most other objects in our scene (other than the loaded OBJ) have computed surface normal's using the cross product.

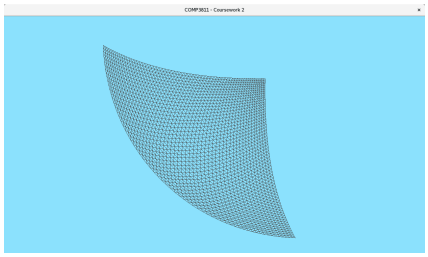

Figure 1: Mesh Render of 1/8 of a sphere

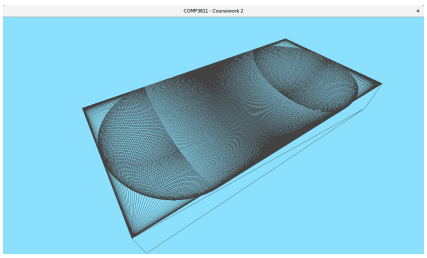

Figure 2: Mesh Render of Our Skate Park Bowl

2.4 Diffuse and Ambient Shading

To implement diffuse and ambient shading we first chose a point to treat as our light source and rendered a cube with pre-calculated normals that we found online to test our implementation. We passed through the light position, light colour, camera position and the cube's plane normal values as uniform values into our shader program. The ambient term was calculated by multiplying our ambient light strength by our light colour. For diffuse, we first calculated the direction the light is travelling by taking the normalised value of the light position minus the fragment position, then calculated the diffuse impact by taking the clamped dot product of the normal vector (also normalised) and the light direction. To get the final diffuse term we multiply the diffuse impact and the light colour. To get our final fragment colour we use: $(ambient + diffuse) * fragBaseColour$.

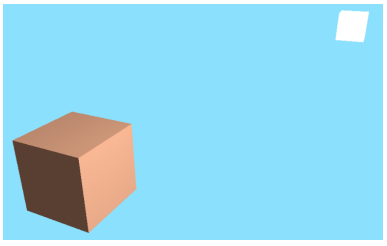

Figure 3: Diffuse and ambient shading shown on a cube

3 Band 50% - 60%

3.1 Animated Objects

We have an animated skateboard rolling up and down our bowl (Figure 11) . This is done by increasing the `angle` variable by the difference in time (divided by two) between the last main loop call and

the current main loop call, making the animation frame rate independent. This is then reversed when the angle reaches a certain value to decrement by the `dt`. The skateboard is then rotated and translated based on this angle. The translations are calculated as follows: `z = ( 10.f * -angle );` and `-y = sqrt( 50 - pow( z, 2 ) ) / 2 - 3.7;`.

3.2 Blinn-Phong Lighting

We have implemented the corrected Blinn-Phong lighting model as described in the lecture slides. To do this we have implemented light and material properties that need to be passed to the shader to compute the output colour. For the light we have 3 properties position, ambience and colour (representing both diffuse and specular). And for the materials, we have ambience, diffuse, specular, shininess and emissive. The lighting values are represented in a struct in and the material values are represented as uniforms in the shader.

When calculating the lighting contribution we first find the distance between the light position and the frag position so that we can implement the inverse square law. We have decided only to apply the inverse square calculation to the colour value of the light (controls the specular and diffuse) as this gained us the best results, we have also applied a scaling factor of 35 to this calculation as we found this to produce the best results in our scene. We calculated the final output colour by calculating the ambient, diffuse and specular values and then summing them together along with the material emissive value. The full formulas for each component we followed from the slides and can be found in our code. When we implemented the multiple light sources we only summed the emissive value once at the end of all the light source calculations.

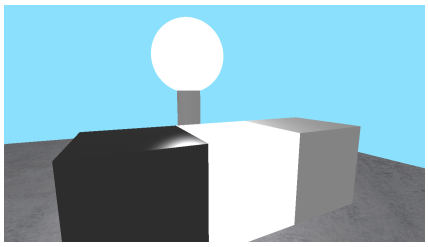

Figure 4: Materials used in our scene (mainly specular, mainly emissive and mainly diffuse

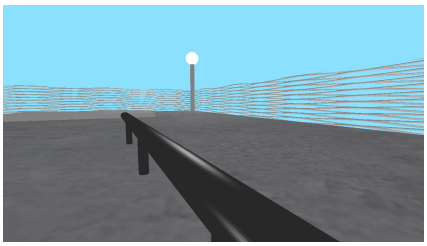

Figure 5: Full corrected Blinn-Phong lighting model shown on a rail in our scene

4 Band 60% - 70%

4.1 Texture Mapping

To create texture mapped objects in our scene we have created a `createTexture(const char* file)` function that takes a file location and creates an appropriate texture name and binds a texture ID to that name. Then we used the `stbi_load` function to load in the texture file and retrieve the height, width and number of colour channels the texture has. We can then move this image data across to the GPU using `gltexImage2d`, this data will be linked with the name and `textureID` we created, therefore when we want to draw to texture onto an object we can call the `glBindTexture` function and pass in the appropriate `textureID`. We have also made sure that we call the `gltexImage2d` with the correct colour channels, if it has 4 channels we call it with `GL_RGBA` and if it has 3 we call it with `GL_RGB`.

To implement most of our texture mapping we have created a `SimpleMeshData` for a basic square tile that includes the correct texture coordinates. We have placed this within a class and created multiple methods to draw other simple objects that only require this tile to be transformed into place and then concatenated. For example with have created textured boxes, textured ramps and textured complex ramps.

Within our fragment shader we needed a way to differentiate from objects that are drawn with texture and ones without textures. Therefore we have passed through a boolean uniform that determines if the texture function is called in the shader or not, we pass this uniform through from the main loop and make sure the right objects are drawn at the correct place in the main loop. Our textures are shown in Figure 6.

4.2 Multiple Light Sources

To create 3 or more light sources in our scene we have set up a light struct that contains values for the light position, ambience and colour as well as making an array of light struct set to be the length of the amount of lights we require in our scene.

We have also constructed a function that will take in the parameters for each light source and compute the resulting contribution each point light has on the scene. This function follows the corrected blinn-phong lighting model which we have highlighted previously. We call this function within a loop in our fragment shader which takes the light properties from each light struct in the light struct array as calculates its effect on the scene. Once we have taken the contribution that each light source, which is summed into a result `vec3` variable we can then add on our material emissive value, which as stated in the lecture slides should be added after all the lights have been calculated.

Then we simply continue like before and either texture the fragment or not, and then make the output colour equal to this resulting `vec3` variable we have just computed.

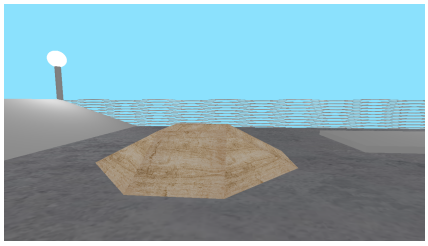

Figure 6: Textures used within our scene

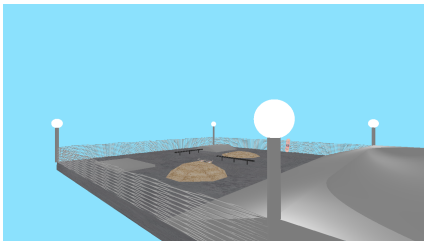

Figure 7: Multiple light sources in our scene

4.3 Loading .OBJ files

We have loaded in a textured skateboard .OBJ file (Figure 9) that we have found online under a free use license for personal use. When we first downloaded it, it didn't have any vertex normals, so we needed to import the OBJ to blender and export again with blenders calculated normals added to the OBJ file. To load this object in we first needed to get the object's normals and texture coordinates. We did this by adding to the implementation of `load_wavefront_obj()` from exercise G4 to use rapidOBJ to parse and load these values, and saved them to a modified version of `SimpleMeshData` from exercise G4 which stores normals and texture coordinates. We changed the texture `v` value to be stored as $1 - v$ as OBJ files index the pixels on screen from the top left. We then created a VAO and drew the object in our scene. We also edited the texture of our object to include our required `markus.png` texture.

4.4 Transparent Objects

For our alpha blended object we have used a texture that contains alpha values where parts of the image are transparent (Figure 8). In line with the theme of our scene this is a metal fence that will outline the edge of our floor. When loading our textures we already loaded in the alpha channel of the image, if there is one, so the alpha values are being passed through to the sharers, all we need to do is enable blending and then specify how we want the blending function to work.

```
glEnable( GL_BLEND );
glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
```

We have enabled blending in OpenGL and then we have set the source factor to alpha value and the destination factor to 1 minus alpha value. This means that the source colour, which will be the transparent part of the rail, will be multiplied by 0 (the alpha value for transparent parts of the image) causing it to have no effect on the outcome colour. Whereas the destination colour, which corresponds to everything on the other side of the rail, will be multiplied by 1 meaning that the final output colour will be entirely determined by the objects and colour on the other side of the railing, hence making it appear transparent. All of the fences are drawn at the end of the main loop so that you can correctly see all of the other objects on the other side of the fence.

5 Band 70% - 100%

5.1 GUI implementation

To implement the GUI we have used `imGui`, where we have included all of the appropriate dependencies within the third_party folder and updated the `premake.lua` appropriately to include this. To open the GUI you just need to press M and then press it again to close the GUI. We have used our GUI to let the user change the light colour of all of the lights in our scene (Figure 10).

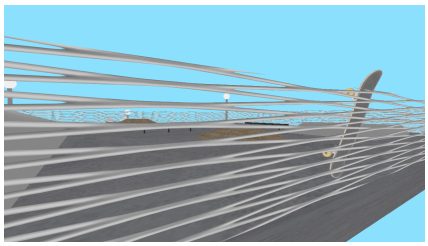

Figure 8: Our transparent fence

Figure 9: Our loaded .OBJ skateboard

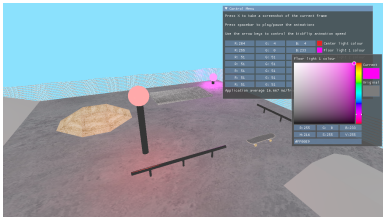

Figure 10: ImGui implemented into our scene

5.2 Screenshots

We implemented a screenshot function in the main file that gets called whenever a user presses the X key. The function creates a vector of chars the same size as the framebuffer ($3 * width * height$) and reads all of the pixels from the framebuffer into it. This vector is then saved as a PNG image using the `stb_image.h` library. The image is stored in the cw2/screenshots folder, where it is named based on how many other screenshots already exist in the folder. A few of the images included above were created using this function.

5.3 Advanced Animation

For our advanced animation we have decided to animate our skateboard model, which has been loaded from an OBJ file. The animation consists of multiple stages, there is linear motion, an ollie, a kickflip (skateboard tricks) and a turn. We have stitched these all together to create a cyclical animation that can be played on repeat, sped up and down as well paused then resumed. Left and right arrow keys control the speed of the animation, and the space bar pauses and resumes the animation.

The animation is frame rate independent. To speed up and slow down our animation we just alter the desired length of the animation. This causes the whereabouts of the skateboard in the animation, so we recalculate the animation time (where in the animation the skateboard is) using the ratio between the length of animation and animation time before changing the length of the animation. Then just multiply this ratio by the length of the animation to get the new animation time. Please see our video for the animation.

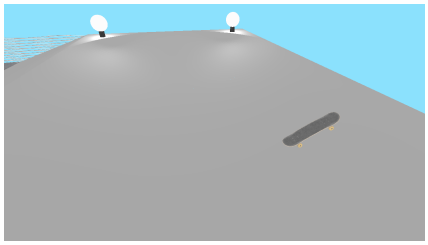

Figure 11: Our basic animation in motion

Figure 12: Our advanced animation in motion

6 Individual Contributions

Task	**Developer**
Vectors and Matrices	Michael (50%), Luke (50%)
Camera	Michael
Complex Objects	Michael (50%), Luke (50%)
Diffuse and Ambient Shading	Michael (50%), Luke (50%)
Animated Objects	Michael (50%), Luke (50%)
Blinn-Phong Lighting	Michael (50%), Luke (50%)
Texture Mapping	Michael
Multiple Light Sources	Michael
.OBJ File	Luke
Transparent Object	Michael
Scene	Michael (50%), Luke (50%)