Computer Graphics Coursework 2

Resizable Screen

To resize the screen properly the viewport which was bound to ‘iwidth’ and ‘iheight’ in the main loop was changed to be bound to ‘nwidth’ and ‘nheight’.

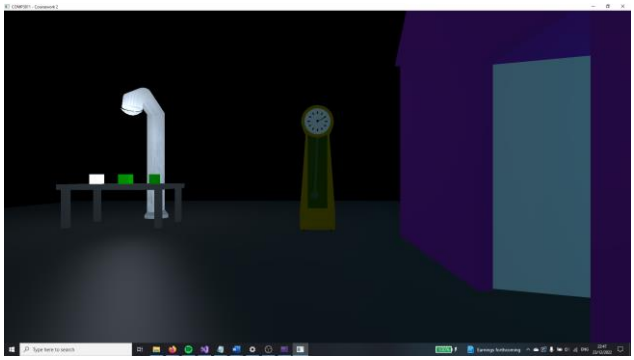

Figure 1: Showing the full screen version of the scene

Camera controlled by movements

To implement the camera, we created a Camera class that contained the camera position, target, and up vectors. It used the UVN camera model system, using WASD to move forward, left, back and right. Q and E are used to move up and down. The mouse inputs are used to change the rotation of the camera. We wanted to implement an FPS style camera, meaning the ‘y’ coordinate of the direction vector is always ‘0’, so, for example if you look UP and press W you won’t go towards where you are looking, but across the XZ plane in the direction you are looking. The camera is frame-rate independent as it uses deltaTime as one of the parameters for calculating movement.

Implementation of the vector matrix classes

We implement same additional function that we thought would be helpful, such as: “deg2rad” so we could do our rotations in degrees, which is more intuitive; “normalise” to vec3 and vec4 to normalise them; “cross” to calculate the cross product of two vectors. Some other operators were also implemented, that were not already in the previous Exercises

Lighting

We implemented the Blinn-Phong shading method. This method uses the equation presented in the slides: $Ip = kaIa + kdId\,(N \cdot L)_{+} + ksIs\,(H \cdot N)_{+}^{\alpha'} + Ie$

Initially in the vertex shader we read in the current position, in world space, and the normalized normal of the of the fragment using:

```
v2fNormal = normalize(uNormalMatrix * iNormal);

ModelPos = (model2world * vec4(iPosition,1.0)).xyz;
```

This information was passed to the vertex shader which also read in the material information and light information.

The code below shows the part of the function in the fragment shader that implements the point lights:

```
vec3 ambient = LightAmbient * materialAmb;
```

```
vec3 normal = normalize(v2fNormal);
vec3 lightDirection = normalize(LightPos - ModelPos);
float diffusion = max( dot(v2fNormal, lightDirection),0.0);
vec3 diffuse = LightDiffse * (diffusion * materialDiff);
vec3 viewDir = normalize(uCameraPos_1 - ModelPos);
vec3 halfwayDir= normalize(lightDirection+viewDir);
float spec = pow(max(dot(v2fNormal,halfwayDir ), 0.0), materialShininess);
vec3 specular = LightSpecular * (spec * materialSpec);

return (diffuse + ambient + specular) * LightColor;
```

The emissive value of the material is then added on afterwards:

```
result+= materialEmissive;
```

where result is the vec3 reading of the point light.

In the scene there are 3 lights: 1 multicoloured light in the lamp in the bedroom, red light from a streetlight and white light from a streetlight.

Figure 2: showing a picture of the multicoloured lamp

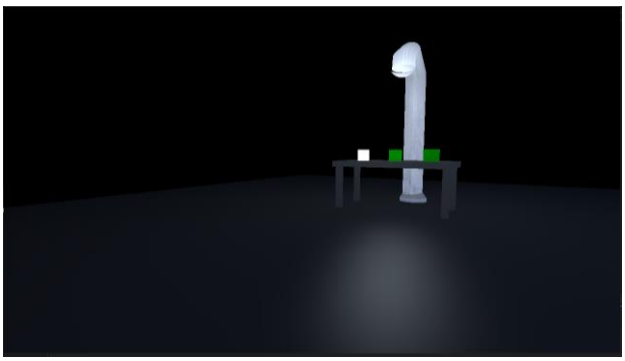

Figure 3:showing the white streetlight

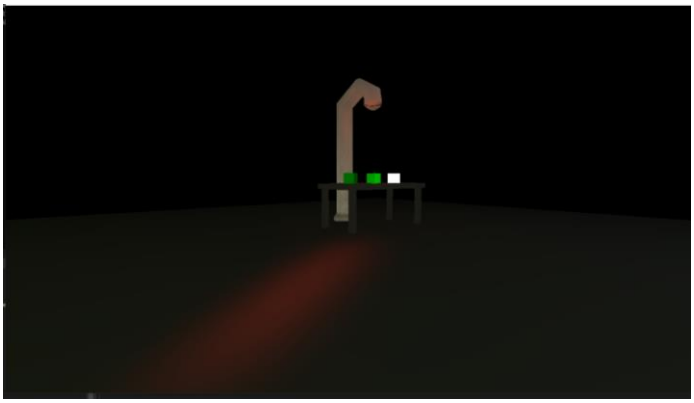

Figure 4:showing the red street light

To see the multicoloured effect see image of the lamp below.

Complex Object

To create the larger complex objects, they were broken down into smaller part which could be created out of the primitive shapes. These smaller shapes where then merged into a single mesh. This allowed for the creation of a grandfather clock and a house as seen below:

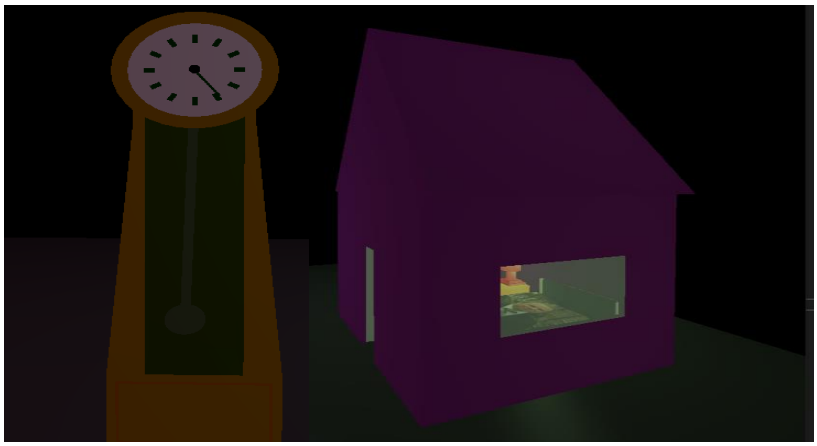

Figure 5:showing the complex structures clock and house

Diffuse, Specular and Emissive Objects

With the Blinn-Phong methods the lighting equation requires the textures of the material. The textures are ambient texture of the light, the diffuse of the material and the specular value of the material and the emissions of the material.

Therefore, to create an emissive object the emissive material of the object is set to a high value. A diffuse object has a high material diffuse value, and the specular object has high specular values. To exaggerate these features the other values were set low:

We represented the diffuse and emissive material as small cubes; these are placed on a table. The specular material is the floor, see figures 3 and 4 .

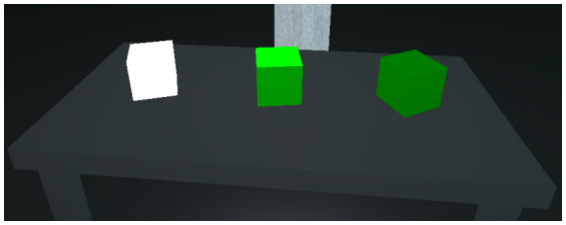

Figure 6:showing the diffuse and emissive objects

Through the images the emissive material is bright and gives off light. The mainly diffuse material has a matte colour, and the specular floor shows reflections of light.

Animation

There are two animations in the scene, the swinging of the pendulum and the ticking of the hands on the clock. To implement these animations so that they are frame rate independent an internal clock is set, and it counts the time between frames and if the animation is not paused it increments the angle if it is paused it does not increment the angle. The pendulum is rotated by:

```
std::sin(angle) * 0.05f
```

```
and the clock face is rotated by:
```

```
make_rotation_x(angle)
```

To implement the speed up and slow down the [and] keys were mapped to a change in speed like the camera controls. The change in position can be seen below:

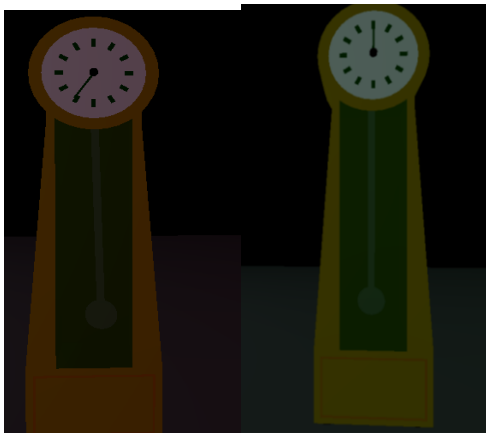

Figure 7: showing the change in position of both clock hand and pendulum

Texture Mapping

Texture mapping allows for textured to be displayed on objects, for example a table looking like it is made of wood. To implement this, we added a vector<Vec2f> to our SimpleMeshData. In the frag shader we got the texture using:

```
vec4 texColor = texture(uTexture, v2fTexCoord);
result2 *= texColor;
```

In main we create texture using:

```
GLuint texture1 =  load_texture_2d("assets/markus.png");
```

And loaded the textures to the frag shader through:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture1);
```

see figure 9 for texture on wavefront objects.

Wavefront Object

A wavefront object is a collection of vertices, normals, texture coordinates and faces. Each have a different notation in the file vertices are noted by 'v' in front and represent the coordinate of each vertex, normals are noted by 'vn' in front, texture coordinates are noted by 'vt' in front and faces are noted by 'f'. The faces are the triangles that make up the object, and each vertex of a triangle is represented by a v/vn/vt [1]. Therefore, the faces are represented in a line as:

f v/vn/vt v/vn/vt v/vn/vt

These can then be passed into a mesh object which when linked to a VAO can be displayed as an object in the scene. We created a few objects for example: a bed with crinkled sheets (achieved through simulations in blender), a bedside table, a bedside lamp, and a streetlight.

Figure 8:wavefront objects

Alpha Blended object

To Implement alpha blending we added an additional parameter to the simple mesh, the alpha filter. This filer would then be passed through the vertex shader and to the fragment shader. Here the oColor would become a 4-float vector, with the additional output being the alpha value of the object. This was used to create a window, see below, and a and a glass cover for the pendulum, see figure 8.

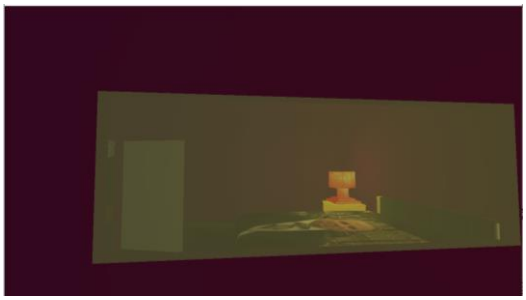

Figure9:window for house

Screenshots

[1] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/

To implement screenshots, we used get read pixels to read the pixels in from the frame buffer. The used the stbi library to flip the pixels as opengl stores images different. Then using the function:

`stbi_write_png`

We wrote to a file name that was created by starting a clock and appending that to the file name.

Figure 10: screenshot

References

1. OpenGlTutorial (no date) *Tutorial 7 : Model Loading*. Available at: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/ (Accessed: December 23, 2022).

Appendix

Task	Contribution
Resizable screen	Tom
Camera Controlled movements	Alex
Lighting	Tom 50% Alex 50%
Complex Objects	Alex
Diffuse, Specular and Emissive objects	Tom
Animation	Alex
Texture Mapping	Alex
Wavefront object creation and implementation	Tom
Alpha blended object	Alex
Screenshot	Alex