CSU44052 Final Project Report

Name:	Anshkirat Singh Dhanju
Student ID:	21355130
Declaration:	I understand that this is an **individual** assessment, and that collaboration is not permitted. I understand that by returning this declaration with my work, I am agreeing with the above statement.
Youtube link 1: Required Features	https://youtu.be/cMjj5RgAKEU
Youtube link 2: Final Demo	https://youtu.be/LV8VKul04Bs

Required feature 1: crowd of animated snow-people/reindeer/robins etc.

Screenshot(s) of feature:

Describe how you implemented it:

First of all, for the purpose of loading model(s), I have used the 'ASSIMP' library. Furthermore, I have also used the 'GLM' library for mathematical computation of matrices which helps us set the position, scale and angle (rotation) of our model.

After loading our model, for the basic animation of our crowd, I consulted the lab 4 solution provided to us. Hence, I proceeded on similar lines decided to use keypress to translate and/or rotate an object of the crowd. To implement this functionality, I created a function 'keyPress()' which was passed as a callback function to 'glutKeyboardFunc()' which is triggered only when a key is pressed. The contents of the 'keyPress()' function include the code for when to translate or rotate an object when a certain key is pressed (shown below in the pseudocode).

Better view of this feature is shown in the video.

Pseudocode:

For model:

```
Variable a;
a = glm :: translate (glm::vec3(coordinates to translate to));
a = glm :: scale(glm::vec3(coordinates to scale));
a = glm :: rotate(glm::radians(angle to rotate to ));
```

For Keypress (movement):

```
Keypress
{
    Switch(key)
            Key 'a' pressed
              Movement in positive x direction += 0.01;
               glutPostRedisplay();
      break;

            Key 'b' pressed
              Movement in positive y direction += 0.01;
               glutPostRedisplay();
      break;

            Key 'c' pressed
              Movement in negative z direction -= 0.01;
               glutPostRedisplay();
      break;
```

```
            Key 'd' pressed
               Rotate anticlockwise += 0.01;
                 glutPostRedisplay();
        break;


}
```

Credits (e.g., list source of any tools, libraries, assets used):

- ASSIMP library – model loading
- GLM library – mathematical computations
- https://learnopengl.com/Model-Loading/Assimp
- https://learnopengl.com/Model-Loading/Mesh
- https://learnopengl.com/Model-Loading/Model
- LAB 4 Solution Provided on Blackboard

Required feature 2: texture-mapping your scene and creatures using an image file

Screenshot(s) of feature:

Few of the **Image Textures** (on left) and **Models** (on right) to depict texture mapping.

Describe how you implemented it:

To implement texture-mapping properly, we use mipmaps. Mipmaps are basically texture images where every subsequent texture is twice smaller than the previous one. They are used to make sure that the texture appears to a viewer depending on its distance. If the viewer is far away, then the mipmap would be small and be hardly visible to the viewer and vice-versa. We simply create a collection of mipmapped textures by calling the function 'glGenerateMipmap()'.

To implement this feature, we used the 'stbi image header' by Sean Barrett, which is a single header image loading library. This is used to load the image we want to use to provide texture to our objects. I have also made use of the 'ASSIMP' library to load models into the scene. The Assimp library returns a pointer which points to the pixel data. This library then helps us extract the texture coordinates, which we then pass into the shader along with the texture as sampler 2D datatype. The texture coordinates are then mapped to the image texture and this is how we give texture to any model.

Pseudocode:

Texture Loading:

```
function get_texture_from_file
{
    load texture using stbi;
    bind texture;
    specify wrapping and filtering of textures;

    return textureID;
}
```

Credits (e.g., list source of any tools, libraries, assets used):

- ASSIMP library – loading model, extract texture coordinates
- STBI_IMAGE_HEADER – loading image texture
- https://learnopengl.com/Getting-started/Textures (mipmaps)
- https://github.com/nothings/stb/blob/master/stb_image.h

Required feature 3: implementation of the Phong Illumination model

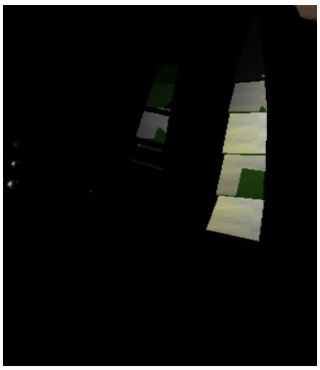
a. *Multiple light sources (at least 2, can be point, directional, or spotlight)*
b. *Multiple different material properties (at least 5 on 5 different objects)*
c. *Normal must be transformed correctly*
d. *Shading must use a combination of ambient, diffuse, and specular lighting*

Screenshot(s) of feature:

Ambient Lighting

Diffuse Lighting

Specular Highlights

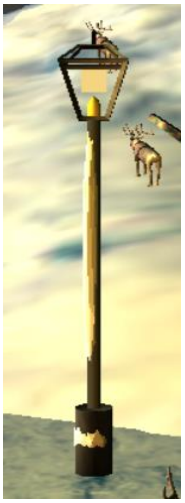


Closer View of Specular Highlight

Overall Lighting of the Scene

Fire Light Source

Lamp Light Source – 1



Lamp Light Source - 2

Describe how you implemented it:

As shown in the images above, I have implemented 4 light sources – one for the whole scene, one for each lamp and one for the fire.

The Phong illumination model is the summation of the ambient, diffuse and specular components of a light source. This has been done in the fragment shader.
To implement the ambient light, we just simply multiply the ambient strength with the light colour.

To implement the diffuse portion of our model, we need the normal of the surface of the object and the light ray direction. We pass the normal from the vertex shader into the fragment shader. Then, we take the dot product of the two. Since, it's a dot product, if the angle between the normal and light ray is 90, there will be no effect of light as the dot product will be zero. Also, the larger the angle between the two, the less effect the light will have on the object. Then finally, we multiple the diffuse value we calculated with the light intensity and light colour.

To implement the specular part of our illumination model, we find the reflection direction using the reflect function. We take light direction to be negative as we want the direction of the reflection and not object to light source direction. Then we find the viewer direction by subtracting the fragment's position (object) from the viewer's position. Next, we find the specular value by taking the dot product of the calculated viewer direction and reflection direction and raise it to power 32 (we can choose any value like 2,4,8,16 etc but I decided to use 32 as it's not too distracting for the eye). We finally multiply the specular value with the light intensity and light colour.

Finally, we just simply add the calculated ambient, diffuse and specular values to complete the implementation of our Phong illumination model.

Pseudocode:

LIGHT:

```
//AMBIENT
ambient = ambientstrength * light colour;

//DIFFUSE
normalize(normal);
light direction = normalise(light position - fragment position);
diffuse factor = dot product(normal and light direction);
diffuse = diffuse factor * light intensity * light colour;

//SPECULAR
viewer direction = normalize(viewer position - fragment position);
reflection direction = reflect(-light direction, normal);
```

```
specular factor = pow(dot product(viewer direction and reflection direction), 32);
specular = specular factor * specular strength * light intensity * light colour;

//TOTAL
total lighting = ambient + diffuse + specular;
```

Credits (e.g., list source of any tools, libraries, assets used):

- Lecture Slides
- https://learnopengl.com/Lighting/Basic-Lighting
- https://riptutorial.com/opengl/example/14741/phong-lighting-model
- http://www.cs.toronto.edu/~jacobson/phong-demo/
- https://www.youtube.com/watch?v=lH61rdpJ5v8

Advanced feature 1

Description/name of feature: **FOG EFFECT**

Screenshot(s) of feature:

Scene Before Fog

Scene After Fog

Describe how you implemented it:

Fog has directly been implemented in the fragment shader.

To implement this feature, first of all, we need the position of objects relative to the camera as the objects closer to the camera would be clearer as opposed to the ones far away from the camera. So, in order to find the relative position matrix, we just simply multiple the world position matrix with the view matrix in the vertex shader and pass it into the fragment shader. We use this to calculate the fog from a depth distance (in position relative to camera or eye coordinates).

Next, I defined a fog colour and a minimum and maximum distance of fog visibility. I am using these 3 parameters to calculate fog factor which gives a value between 0 and 1 and tells us about the intensity of the fog. To get the fog factor between 0 and 1, I am using the 'clamp()' function which keeps the resultant fog factor between the desired values which in our case are 0 and 1.

Lastly, I am using the 'mix()' function to mix the fragment colour(overall scene and the objects in it) with the fog colour to get a smooth blend of both.

Pseudocode:

FOG EFFECT:

```
position relative to the camera = view matrix * world position matrix;
pos = position relative to the camera;
fog factor = (maximum distance - pos distanc4) / (maximum distance - minimum distance);
fog factor = clamp (fog factor between 0 and 1);
overall colour = mix(fog colour and fragment colour);
```

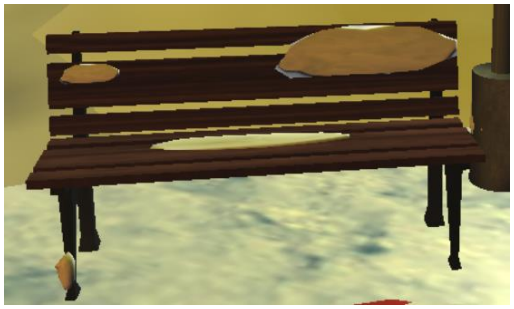
Credits (e.g., list source of any tools, libraries, assets used):

- https://moddb.fandom.com/wiki/OpenGL:Tutorials:Tutorial_Framework:Light_and_Fog
- https://opengl-notes.readthedocs.io/en/latest/topics/texturing/aliasing.html
- https://www.mbsoftworks.sk/tutorials/opengl4/020-fog/
- https://www.youtube.com/watch?v=qslBNLeSPUc&list=PLRIWtICgwaX0u7Rf9zkZhLoLuZVfUksDP&index=16

Advanced feature 2

Description/name of feature: **SELF MADE MODELS**

Screenshot(s) of feature:



Describe how you implemented it:

I used the 'Blender' software to create and modify the models used in the scene. I used modelling, texture painting and other relevant tools to make/modify the models used in the scene. For the purpose of texture painting, I have used image textures (to add texture to the models).

The **snowman, bench, car, lamp post, the chair**, **snowflake** and the **fire outside the house** have been completely self-made.

The models of **trees** and **cottage** have all been modified to suit the winter theme of our project by adding a snowy component to them and adding texture to models without them.

Only the **reindeer, terrain, barrier** and the **road** are models which have been loaded from the resources mentioned below and loaded to the scene as it is.

Pseudocode:

.blend files provided in the submission.

Credits (e.g., list source of any tools, libraries, assets used):

- Blender 3.0
- https://free3d.com/
- https://www.cgtrader.com/free-3d-models (Majority of premade models taken from here and then edited in blender)
- https://sketchfab.com/features/free-3d-models

Advanced feature 3

Description/name of feature: **CUBE MAPPING – SKY-BOX**

Screenshot(s) of feature:

Scene without Sky Box

Scene with Sky Box

Describe how you implemented it:

Cubemap is a texture that contains 6 2D textures that forms a textured cube. Cube map gives us the impression of being part a compact realistic environment. I have implemented the cube maps in the main.cpp file. Since, cubemap is also a texture, we use the 'glBindTexture()' to generate a texture and bind it. We are also using the 'stbi image header' to load image textures.

To implement the skybox, I have created a function 'loadskymap()' which is loading the textures required for our skybox. For displaying the sky map, I have created a different set of vertex and fragment shader – 'box_shader.vert' and 'box_fragment.frag' respectively.

Pseudocode:

SKYBOX:

```
faces = contains image texture path;
function loadskymap (faces)
{
   loading image texture using stbi;
   specify wrapping and filtering methods of texture;
   return textureID;
}
```

Credits (e.g., list source of any tools, libraries, assets used):

- STBI IMAGE HEADER – to load image header
- https://learnopengl.com/Advanced-OpenGL/Cubemaps
- https://www.youtube.com/watch?v=_Ix5oN8eC1E&list=PLRIWtICgwaX0u7Rf9zkZhLoLuZVfUksDP&index=27
- https://www.youtube.com/watch?v=8sVvxeKI9Pk
- https://opengameart.org/content/winter-skyboxes (skybox image texture assets)

Advanced feature 4

Description/name of feature: **PARTICLE EFFECT (SNOWING EFFECT)**

Screenshot(s) of feature:

Scene without Snowflakes

Scene with Snowflakes Falling

Describe how you implemented it:

To implement the particle effect and show the effect of snow, fist we need to create particles or the snow particles in our case. To do this, firstly, I am declaring the maximum number of particles that will be falling from the sky. Depending upon the complexity of the model one is using, the number of particles could slow down the program. Then in the main, I am initializing all the particles by calling the function 'createParticles()' the same number of times, the number of particles we have created. This function includes the creation of particle, its lifetime and decay rate. Also, the function contains information regarding the velocity of fall of the snow and the gravitational resistance it will face.

We are displaying the particle effect simply by calling the function 'drawSnow()' in the 'display()' function of our code. In this function, firstly I am loading the model I want as the snow particle which in my case is a self-made snowflake created using blender. Then we are simply just moving the particle from sky towards the ground and decaying it at the same time. We call the 'createParticle()' again in case the life of the particles in the system is over. This gives us the impression that the snow is falling continuously.

Pseudocode:

SNOWFLAKES FALLING DOWN:

```
define number of particles;

function to initialize particles
{
   particle state; (alive or not)
   particle life;
   particle fade;
   particle spread;(across x and y axis);
   particle falling height (along y);
   particle falling velocity;
   particle resistance (gravity);
}

function to draw snowflakes
{
   position of particles in x,y,z;
   importing model to be used as the particle;
```

```
  falling of snowflake via translation from sky;
  decay of snowflake;
  calling initiation function again to revive the snowflakes;
}
```

Credits (e.g., list source of any tools, libraries, assets used):

- https://learnopengl.com/In-Practice/2D-Game/Particles
- https://www.youtube.com/watch?v=6PkjU9LaDTQ&list=PLRIWtICgwaX0u7Rf9zkZhLoLuZVfUksDP&index=34
- https://www.mbsoftworks.sk/tutorials/opengl3/23-particle-system/
- https://users.soe.ucsc.edu/~pang/161/w09/submit/projects/mang/finalproject.html