Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

I have read and I understand the plagiarism provisions in the General Regulations of the University

Student Name	Tianze Zhang
Student ID Number	19334401
Course Title	Computer Science
Module Title	CSU44052: Computer Graphics
Lecturer(s)	Prof. Rachel McDonnell Prof. Carol O'Sullivan
Title	Computer Graphics – Final Report
Youtube link 1: Required Features	https://youtu.be/srMH6e6A6bo
Youtube link 2: Final Demo	https://youtu.be/LCakMi2JI54

1 Required Features

1.1 Crowd of animated snow-people/reindeer/robins etc.

1.1.1 Screenshot(s) of feature:

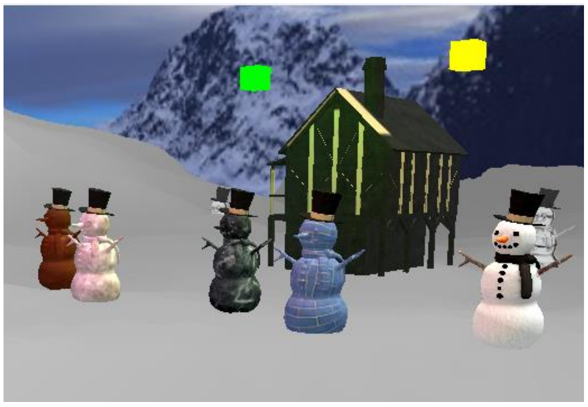


Figure 1.0: A animated snow-people crowed which heads are shaking and moving back and forwards.

1.1.2 implementation:

The animation of each individual snowman is shaking its head. This feature was implemented by splitting the head and body of the snowman model into two separate object model. This was achieved using the blender bisect cutting feature where a plane was inserted between the snowman's head and the snowman's body, and only keeping the upper/lower part of the plane. Both upper and lower part were exported as a .obj file which was then loaded to are openGL program using assimp. The head-shaking animation was created by adding a constant the rotation matrix of the head model over time, checking the condition that it was not over certain degree which leads to turning the head entirely around. If it's over the conditioned degree, then the rotation constant was minus instead. The animation of the crowed is moving backwards and forwards, this was achieved similarly to the headshaking feature, but this time adding a constant to their x axis of their translation matrix. Both those moving backward and forward had a boundary set which they can't go over, if they go over the constant will be subtracted instead and vice versa.

1.1.3 Pseudocode:

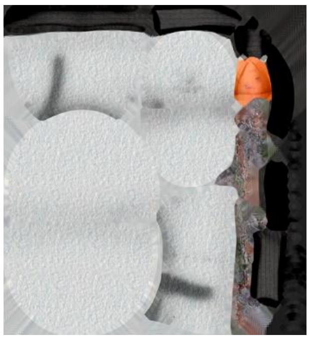
```
if (crntTime - prevTime >= 1 / 60)
{
    if (move_forward <= forward_boundary && doMove) {
        move_forward -= constant;
        move_backward += constant;
        prevTime = crntTime;
        if (move_forward < boundary) {
            doMove = false;
        }
    }
    else {
        move_forward += constant;
        move_backward -= constant;
        prevTime = crntTime;
        if (move_forward >= 57.0f) {
            doMove = true;
        }
    }

}
```

1.1.4 Credits

blender (Blender, 2023), assimp (assimp, 2023), sketchfab (Sketchfab, 2023).

1.2 texture-mapping your scene and creatures using an image file

1.2.1 Screenshot(s) of feature:



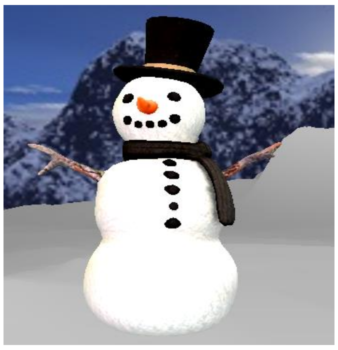


Figure 1.1: The snowman's texture's image png before mapping,

Figure 1,2: The snowman's texture after UV mapping.

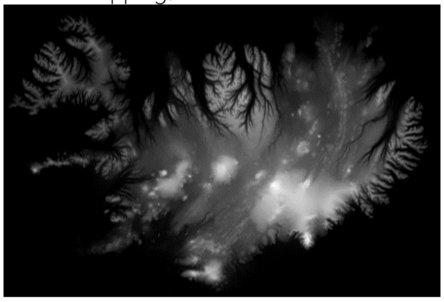


Figure 1.3: The terrain's texture's image png before mapping

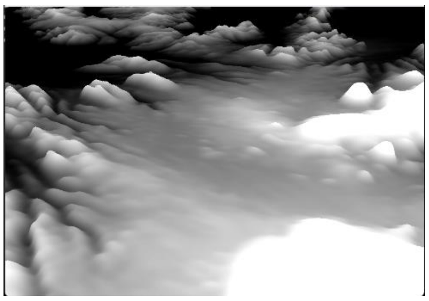


Figure 1.4: the terrian's texture after uv mapping.

1.2.2 implementation:

I provided two ways of binding texture, for the models since most are models that were downloaded from sketchfab. Created or cut by me using blender. Therefore, most of the texture mapping was embedded in the exported .obj file. Texture mapping involves telling which section of the given texture corresponds to which texture co-coordinates. Those data were stored in a .mtl file along with the .obj after exporting from blender. When loading models to openGL, assimp automatically reads in those data and stored in the mesh class. For images not banded to models on blender, I have also implemented a load_texture() function that takes in a path to the texture image, do texture wrapping and filtering techniques using glTextParamteri(), the wrapping method that was chosen is repeat and linear for the filtering method. The function returns an unsigned integer linking to the texture. Texture were bind to objects using glActiveTexture() and glBindTexture() in later stage. For this project, the heightmap, diffuse and specular map of objects were bind using this method.

1.2.3 Pseudocode:

```
unsigned int loadTexture(char const* path)
{
    …
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);

    return textureID;
}

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, id);
```

1.2.4 Credits

Learnopengl textures article (OpenGL, Textures, 2023).
FILEFormat obj (FILEFORMAT, 2023).
Blender (Blender, 2023).

1.3 implementation of the Phong Illumination model

1.3.1 Screenshot(s) of feature:

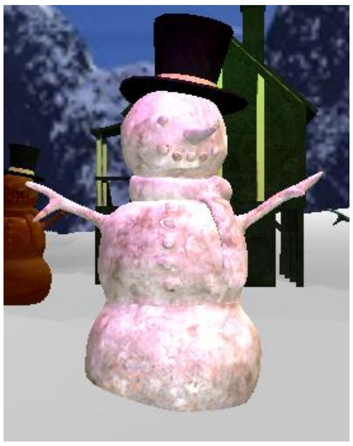


Figure 1.5: example of snowman, with rusty metallic materials under purple light. Reflects more light than 1.6.

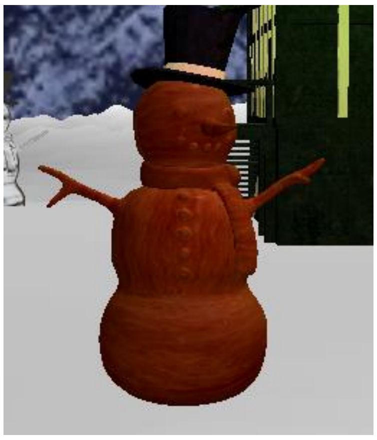


Figure 1.6: example of snowman, with wood materials under purple light, reflect less light than 1.5.

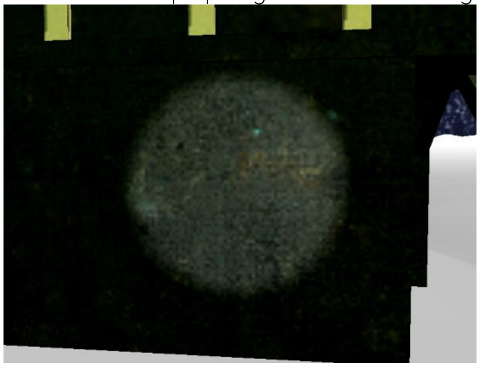


Figure 1.7: example of a spotlight.

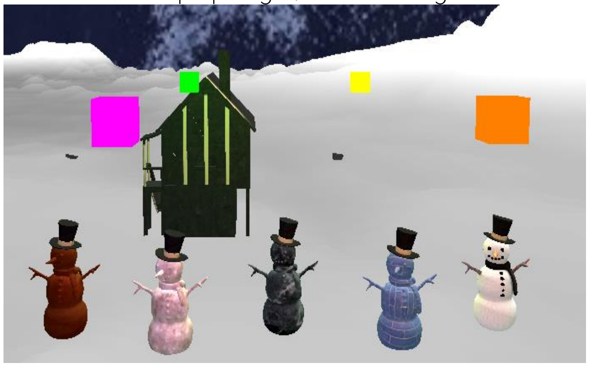


Figure 1.9: example of multiple coloured point light source, with 5 different snowman material properties including (left to right): wood, metal, marble, tiles and snow.

1.3.2 implementation:

Part (a): This project consist of three types of light sources and 6 lights in total. There are 4 point lights, 1 spotlight and 1 directional light. Point light are lights from a light source, those point light sources are implemented as coloured cubes, which the colour of the cube represents the colour of the light. The directional light is coming from x=-0.2, y=-0.1 and z=-0.3 which represents the light coming from the sum. The spotlight is coming from the camera position, this assumes that the character is carrying a light source like a torch.

Part(b): There are 5 different objects with 5 properties as in Figure1.9, with wood, metal, marble and snow. I chose to implement the material effect (How much light it takes) in the fragment shader. The material property is defined as a struct which consist of diffuse, specular and shininess. Diffuse define the colour of surface under diffuse lighting, specular define the colour of the surface under diffuse lighting and shininess defines the shape of the light. Different colour will have its own material diffuse and specular.

Part(c): The normal is calculated correctly as follows: assimps first loads and calculated the tangents and bitangents from the normal map. Those were then sent to the vertex shader to calculate the tangent space and finally sent to the fragment shader to calculate the normal.

Part(d): The final colour of each light source it calculated by the combination of it's ambient, specular and diffuse shading multiplied by the material properties. Each of the light sources final colour was combined together which determined the final colour of each frame. Those final colour were outputted from the fragment shader.

1.3.3 Pseudocode:

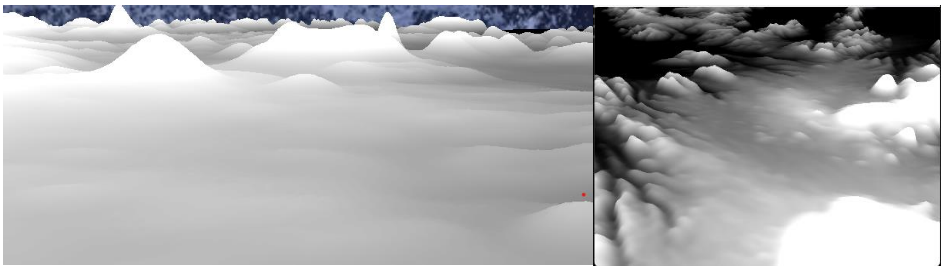
```
void main()
{
    vec3 result = CalcDirLight(dirLight, norm, viewDir);
    result += CalcPointLight(pointLight, norm, FragPos, viewDir);
    result += CalcSpotLight(spotLight, norm, FragPos, viewDir);

    FragColor = vec4(result, 1.0)
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
    …
    vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, 
TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, 
TexCoords));
    …
    return (ambient + diffuse + specular);
}
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)

vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
```

1.3.4 Credits

Learn OpenGL website lighting chapter (OpenGL, Multiple lights, 2023)
Fundamentals of Computer Graphics, 4th Edition (Steve Marschner, 2018)

2 Advanced Features

2.1 Height-mapped terrain


Picture 2.0: height map

2.1.1 implementation:

To render the terrain, height map is applied to the mesh. Height map is a greyscale image where the texture value corresponding to the height value of the terrain at the point. To create 3D terrain, heightmap is often used. Render the terrain using CPU static method, which calculates once. When rendering the terrain, the y of vertex in the fragment shader was normalize to a grayscale value. To get high quality terrain, the high-resolution mesh generation matches the resolution of image.

2.1.2 Pseudocode:

```
// render the mesh triangle strip by triangle strip - each row at a time
for (unsigned strip = 0; strip < numStrips; strip++)
```

```
{
    glDrawElements(GL_TRIANGLE_STRIP,    // primitive type
    numTrisPerStrip + 2,   // number of indices to render
    GL_UNSIGNED_INT,     // index data type
    (void*)(sizeof(unsigned) * (numTrisPerStrip + 2) * strip)); // offset to
starting index
}
```

2.1.3 Credits

Learn Opengl heightmaps (OpenGL, Tessellation Chapter I: Rendering Terrain using Height Maps, 2023).
Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion. (Vries, 2020)

2.2 Smoke of fog

2.2.1 Screenshot(s) of feature:

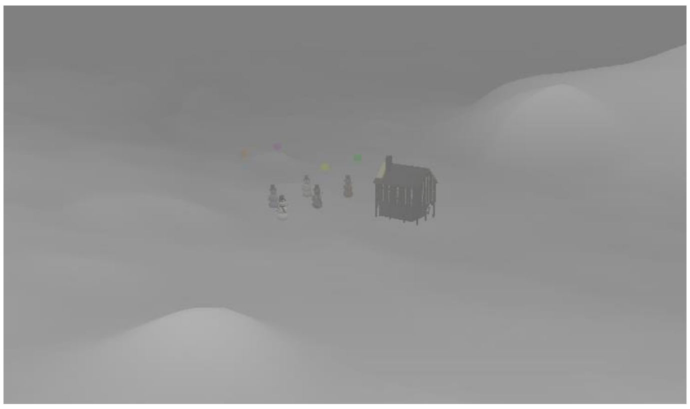


Figure 2.2: foggy feature.

2.2.2 implementation:

The fog was implemented in the vertex and fragment shader. Both exponential and linear fogging was implemented in this project, however, exponential fogging was mostly used. In each shader, this output colour this time is determined by a mixture of its original colour, the fog colour and the fog factor. The fog factor is a coefficient that is determined by the distance from the camera to the object, the longer the distance the stronger the coefficient hence stronger fog. The distance is calculated in the vertex shader. In exponential fogging the coefficient is calculated by: `exp(-pow(params.density * fogCoordinate, 2.0));`

2.2.3 Pseudocode:

```
void main()
{
    if(fogParams.isEnabled)
    {
          float fogCoordinate = abs(ioEyeSpacePosition.z / ioEyeSpacePosition.w);
          FragColor = mix(FragColor, vec4(fogParams.color, 1.0),
getFogFactor(fogParams,fogCoordinate));
    }
}
```

2.2.4 Credits

MBsoftworks (Softworks, 2023)

2.3 Partially transparent geometry using alpha-blending

2.3.1 Screenshot(s) of feature:

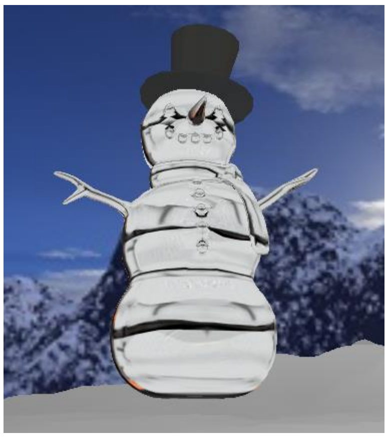


Figure 2.2: semi-transparent snowman shows portions the colours of mountain

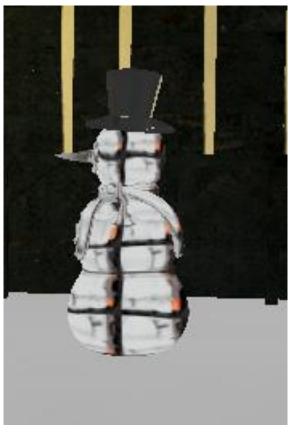


Figure2.4: shows portions of the house's stick

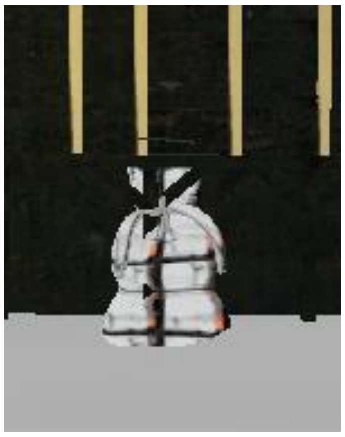


Figure 2.5: object fading due to alpha < 0.1

2.3.2 implementation:

To handle transparency within object, blending in the fragment shader is required for the transparent objects in the final stage of the pipeline. Blending means combining the colours from transparent object itself with all other objects. During blending stage, the current primitive colour created from the fragment shader presented in the framebuffer is combined with the colours of the other objects in some suitable ways by the blending function. Fragment alpha value, which represents the opacity of the object, is used to control the blending of colours. Enable blending in OpenGL, calling glEnable with GL_BLEND. Also, to get the blending result of our little two square example, we want to take the alpha of the source colour vector for the source factor and 1−alpha of the same colour vector for the destination factor. This was done by glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); Alpha value lower than a threshold of 0.1, discard the fragment, no transparency.

2.3.3 Pseudocode:

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

2.3.4 Credits

Learn OpenGL blending chapter (LearnOpenGL, Blending, 2023)
Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion. (Vries, 2020)

2.4 Cube maps/Skybox

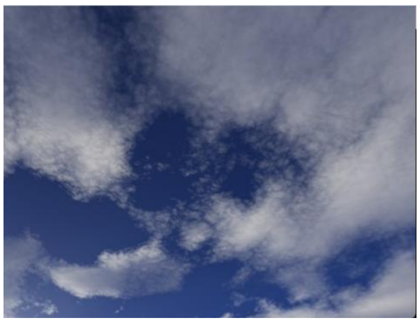
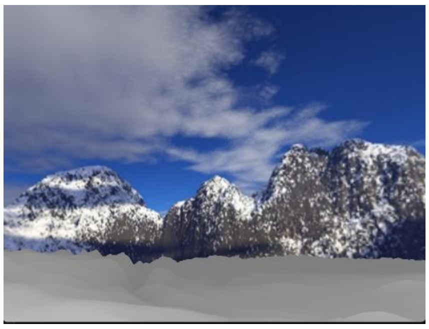



Figure 2.6 & 2.7: skyboxes

2.4.1 implementation:

Cubemap is a texture of containing six 2D square/face textures of a cube that can be used to sample the texture value. It is used to represent the environment in 3D graphics, and for tasks of skyboxes and environment mapping. Creating Skybox cube includes 6 images/6faces of an environment. When texturing 3D cube, the skybox was sampled and use the position vector get the texture value. the skybox was then loaded as a cubemap with cubemap texture. For vertex shader, set incoming local position vector as the outcoming texture coordinates for the fragment shader.fragment shader takes vertex's interpolated. Finally, position vector to sample the texture values from cube map, and then skybox is rendered. The skybox was rendered last, this reduce the fragment shader calls, provides performance improvement.

2.4.2 Pseudocode:

```
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
```

2.4.3 Credits

LearnOpenGL cubemaps (LearnOpenGL, Cubemaps, 2023)
Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion. (Vries, 2020)

2.5 More advanced texturing effects: specular mapping

2.5.1 Screenshot(s) of feature:

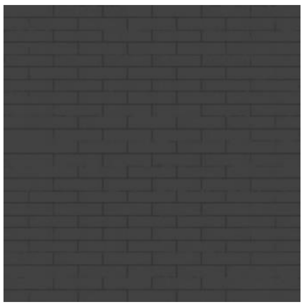
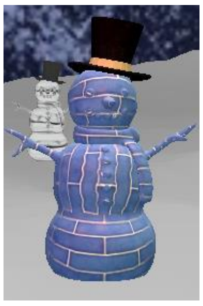



Figure 2.8: diffuse map Figure 2.9: specular map Figure 3.0: model diffuse and specular mapping

2.5.2 implementation:

The diffuse mapping image is basically the objects texture, while specular mapping image is used to define the shininess/highlight on a surface. For example, in the tiles image in figure 2.8, the white colour between the tiles should have more highlights than tiles itself. Therefore, it's specular map should highlight them more, marking them darker in figure 2.9. This result the white part in figure 3.0 having more lights colour reflected than the blue colour. To implement this in openGL, simply the material diffuse and material specular were changed from a vec3 to a sampler2D which takes in an image. Textures were simply loaded and added as the same load_texture() function discussed in previous section, and the shader will then use those colour to calculate the final colour.

2.5.3 Pseudocode:

```
struct Material {
    sampler2D diffuse;
    sampler2D specular;
    float shininess;
};
```

2.5.4 Credits

OpenGL lighting Chapter (OpenGL, Multiple lights, 2023)
Poliigon (POLLIGON, 2023)
Fundamentals of Computer Graphics, 4th Edition (Steve Marschner, 2018)

3 References

assimp. (2023, January 6). assimp. Retrieved from Github: https://github.com/assimp/assimp
Blender. (2023, January 6). Blender. Retrieved from Blender: https://www.blender.org/
FILEFORMAT. (2023, January 6). What is an OBJ File? Retrieved from FILEFORMAT: https://docs.fileformat.com/3d/obj/
LearnOpenGL. (2023, January 6). Blending. Retrieved from LearnOpenGL: https://learnopengl.com/Advanced-OpenGL/Blending
LearnOpenGL. (2023, January 6). Cubemaps. Retrieved from LearnOpenGL: https://learnopengl.com/Advanced-OpenGL/Cubemaps
OpenGL, L. (2023, January 6). Multiple lights. Retrieved from Learn OpenGL: https://learnopengl.com/Lighting/Multiple-lights
OpenGL, L. (2023, January 6). Tessellation Chapter I: Rendering Terrain using Height Maps. Retrieved from Learn OpenGL: https://learnopengl.com/Guest-Articles/2021/Tessellation/Height-map
OpenGL, L. (2023, January 6). Textures. Retrieved from Learn OpenGL: https://learnopengl.com/Getting-started/Textures
POLLIGON. (2023, January 6). Tectures. Retrieved from POLLIGON: https://www.poliigon.com/textures/bricks
Sketchfab. (2023, January 6). Sketchfab. Retrieved from Sketchfab: https://sketchfab.com/feed
Softworks, M. (2023, January 6). Fog basics. Retrieved from MegaByte Softworks: https://www.mbsoftworks.sk/tutorials/opengl4/020-fog/
Steve Marschner, P. S. (2018). Fundamentals of Computer Graphics, 4th Edition. A K Peters/CRC Press.
Vries, J. d. (2020). Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion. KW.