TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING PULCHOWK CAMPUS

SECOND YEAR SECOND PART(II/II)

A PROJECT REPORT ON

OPENGL BASED GAME ENGINE

SUBMITTED BY:

PRASHANT SHRESTHA PUL075BEI024

SIMON G.C. PUL075BEI036

SUBASH MAINALI PUL075BEI039

UDAYA RAJ SUBEDI PUL075BEI047

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

December 2020

OPENGL BASED

GAME ENGINE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE COURSE OF SECOND II/II PART DATA STRUCTURE AND ALGORITHM

BACHELOR OF ELECTRONICS, COMMUNICATION AND INFORMATION ENGINEERING

SUBMITTED BY:

SUBMITTED BY:

PRASHANT SHRESTHA PUL075BEI024

SIMON G.C. PUL075BEI036

SUBASH MAINALI PUL075BEI039

UDAYA RAJ SUBEDI PUL075BEI047

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

December 2020

Acknowledgement

We would like to express our sincere gratitude to lecturer Mr. Anil Verma sir, theory lecturer on Computer Graphics Pulchowk Campus, Lalitpur for this immense opportunity and for his constant guidance and support in our lab work. We were able to complete this project on time only because we had a clear understanding of the computer graphics concepts taught in theory and lab classes. We believe this project has played a pivotal role in sharpening our programming and problem-solving skills using graphics concepts.

Besides, we would like to appreciate the constant help and support of our classmates whenever we had any issue. Their suggestion kept us motivated to keep trying and make it happen. On the other hand, thanks to everyone who crossed questioned our ideas and compelled us to come up with a better idea which eventually helped to make our work better.

Every one of us in the team have given their personal best. We believe this project will play a key role in our future endeavors.

Prashant Shrestha (BEI024)

Simon G.C (BEI036)

Subash Mainali (BEI039)

Udaya Raj Subedi (BEI047)

Contents

Introduction 1

Problem Statement 3

Objectives 3

Project Description 4

- OpenGL Pipeline: 4
  - Vertex Shader 5
  - Shape Assembly 6
  - Geometry Shader 6
  - Rasterization 6
  - Fragment Shader 6
  - Per-Sample Operations 6
  - Transformation: 7
  - GLM 7
  - Scaling: 7
  - Translation: 7
- Camera: 8
- Projection: 10
- Rendering workflow: 11
- Lighting 13
  - Ambient lighting: 13
  - Diffuse lighting: 14
  - Specular lighting: 15
- Textures 17
  - Texture Coordinates: 17
  - Sampling and Wrapping 19
  - Filtering and Mipmap 19
  - Texture Units 21

Game play 22

- Player Movement: 22
- Enemy Movement 22

Final Analysis 24

- Result 24
- Application 25
- Limitation 26
- Problem Faced 26

Conclusion 27

Source Code 28

Bibliography 42

List of Figures

Figure 1 Complete rendering pipeline of OpenGL....4
Figure 2 Local coordinate system of camera aimed at a point....8
Figure 3 Computing axes of the camera....9
Figure 4 Look At matrix....10
Figure 5 Perspective projection frustum....11
Figure 1: Components of Phong Lighting Model....13
Figure 2: Ambient Lighting....13
Figure 3: Effect of diffused lighting....14
Figure 4: Diffused Lighting....14
Figure 5 Specular Lighting....15
Figure 6: Effect of specular lighting....15
Figure 7: Result of specular lighting for different values of power term....16
Figure 8: Final Result after adding all components of phong model....16
Figure 9: Cube with texture....17
Figure 10: Colourful cube....17
Figure 11: Texture coordinate....18
Figure 12: Texture coordinate2....18
Figure 13: Texture Coordinate, a vertex attribute....18
Figure 14: GL_LINEAR....20
Figure 15: GL_NEAREST....20
Figure 16: Wrapping and filter setting....20
Figure 17: Mipmap....20
Figure 18: Mipmap process....21
Figure 19: Texture Units....21
Figure 20 : Player's view....23
Figure 21: Rendered Map....23
Figure 22: Game play scene....23
Figure 23: Final Game Scene....24
Figure 24: Enemy chasing the player....25

Introduction

Computer graphics studies the manipulation of visual and geometric information using computational algorithms and techniques. It focuses on the mathematical and computational foundations of image generation and processing. Computer graphics is a sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content. Although the term often refers to the study of three-dimensional computer graphics, it also encompasses two-dimensional graphics and image processing. Computer Graphics is used where a set of images needs to be manipulated or the creation of the image in the form of pixels and is drawn on the computer. Computer Graphics can be used in digital photography, film, entertainment, electronic gadgets and all other core technologies which are required. It is a vast subject and area in the field of computer science. Computer Graphics can be used in UI design, rendering, geometric objects, animation and many more. In most areas, computer graphics is an abbreviation of CG. There are several tools used for implementation of Computer Graphics. In our project however we have used OpenGL and its GLEW library to accomplish desired graphical output of our project.

OpenGL (Open Graphics Library) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.

Silicon Graphics, Inc. (SGI) began developing OpenGL in 1991 and released it on June 30, 1992 applications use it extensively in the fields of computer-aided design (CAD), virtual reality, scientific visualization, information visualization, flight simulation, and video games. Since 2006, OpenGL has been managed by the non-profit technology consortium Khronos Group. OpenGL is an evolving API. New versions of the OpenGL specifications are regularly released by the Khronos Group, each of which extends the API to support various new features. The details of each version are decided by consensus between the Group's members, including graphics card manufacturers, operating system designers, and general technology companies such as Mozilla and Google.

We decided to use OpenGL as it has cross platform support and it has a good community and enough documentation for us to follow and learn properly about computer graphics. Also, it met the requirements we need for the easy accomplishment of the end result we desired.

Our project for computer graphics is a game engine which generates a map where the different objects of the game i.e., player, enemy, obstacle (wood box), wall, light source, floor are generated from a 2D array where the place value of different objects are stored and rendered accordingly in the world. The theme of the game is the player is constantly being chased by the enemy which has pathfinding algorithm implemented (A*) and has to reach the goal by navigating through maze-like map structures rendered in the game.

The objects in the game have been rendered with the help of vertex arrays which are loaded and mapped accordingly depending on the texture we want to use. The global illumination of the scene in the game is achieved through lighting whose ambience, diffusion and specular attributes can be controlled as per the user needs. The camera movement is free to look around all the axis and can navigate through the map via the use of mouse and W, A, S, D keys of the keyboard.

Problem Statement

1. Given a 2D occupancy grid, we need to generate a map for the game object to navigate around and render it.
2. Lighting: after the scene is rendered, we need to have illumination for the object rendered where they will have different parameters for ambience, diffusion, specular reflection in order to mimic real world object texture and lighting situations.

Objectives

1. Implementation scene and texture rendering
2. Study and implementation of lighting and shading.
3. Implementation of 2D and 3D transformation.
4. Implementation of perspective projection in computer graphics.
5. Implementation of vertex buffer for texture mapping and model rendering.

Project Description

OpenGL Pipeline:

In OpenGL everything is in 3D space, but the screen or window is a 2D array of pixels so a large part of OpenGL's work is about transforming all 3D coordinates to 2D pixels that fit on your screen. The process of transforming 3D coordinates to 2D pixels is managed by the graphics pipeline of OpenGL. The graphics pipeline can be divided into two large parts: the first transforms your 3D coordinates into 2D coordinates and the second part transforms the 2D coordinates into actual-colored pixels.

The graphics pipeline can be divided into several steps where each step requires the output of the previous step as its input. All of these steps are highly specialized (they have one specific function) and can easily be executed in parallel. The processing cores run small programs on the GPU for each step of the pipeline. These small programs are called shaders.

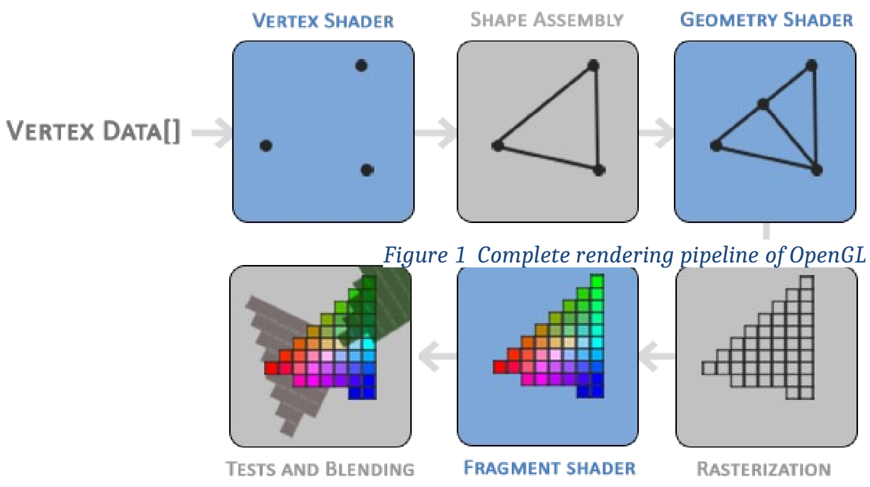
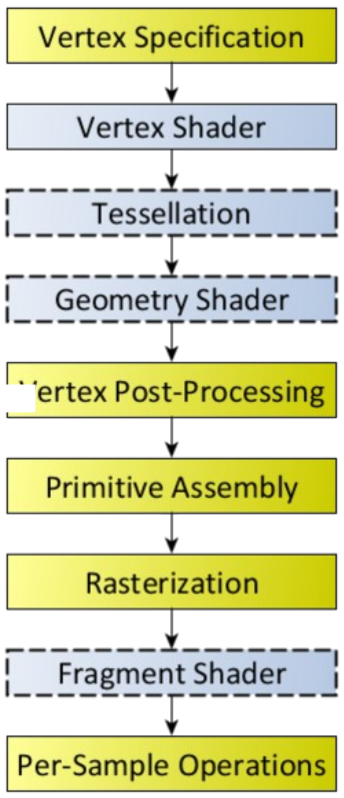
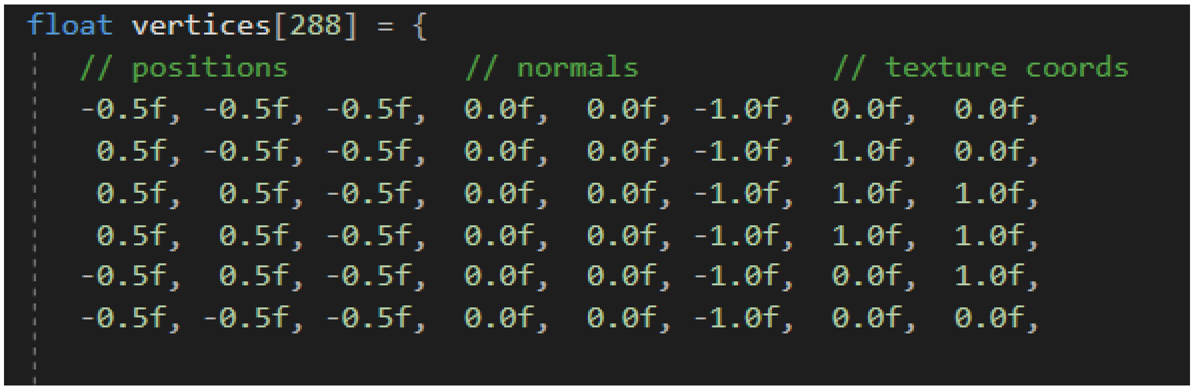


Figure 1 Complete rendering pipeline of OpenGL



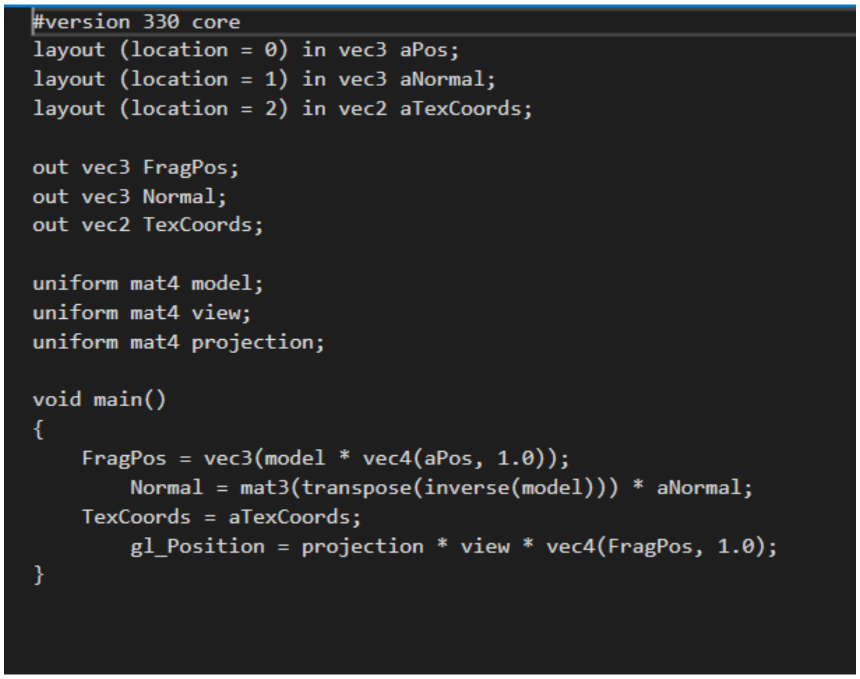
As input to the graphics pipeline, we pass in a list of three 3D coordinates that should form a triangle in an array here called Vertex Data; this vertex data is a collection of vertices. A vertex is a collection of data per 3D coordinate. Vertex doesn't mean only the 3D- position but also the texture coordinates, normal, tangents and many others. This vertex's data is represented using vertex attributes that can contain any data.



```
float vertices[288] = {
    // positions          // normals           // texture coords
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  0.0f,
     0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  0.0f,
     0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  1.0f,
     0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  1.0f,
    -0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  1.0f,
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  0.0f,
```

Vertex Shader

The main purpose of the vertex shader is to transform 3D coordinates to post-projection space, for consumption by the Vertex Post-Processing stage (Clipping and Transformation), and the vertex shader allows us to do some basic processing on the vertex attributes.



```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    FragPos = vec3(model * vec4(aPos, 1.0));
        Normal = mat3(transpose(inverse(model))) * aNormal;
    TexCoords = aTexCoords;
        gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

Shape Assembly

In Graphics, many of the objects are formed using triangles because it makes easy to manipulate and keep track of different parameters. This job is done using Shape Assembly. The primitive assembly stage takes as input all the vertices from the vertex shader that form a primitive and assembles all the point(s) in the primitive shape

Geometry Shader

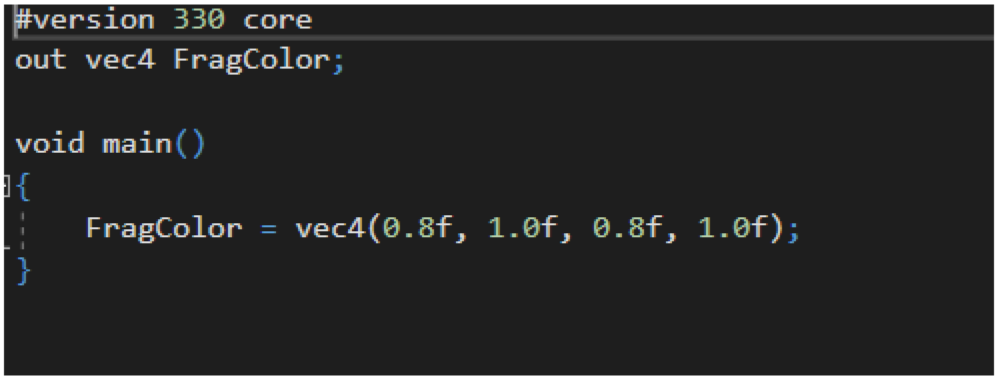
The output of the primitive assembly stage is passed to the geometry shader. The geometry shader takes as input a collection of vertices that form a primitive and has the ability to generate other shapes by emitting new vertices to form primitives.

Rasterization

The output of the geometry shader is then passed on to the rasterization stage where it maps the resulting primitive(s) to the corresponding pixels on the final screen, resulting in fragments for the fragment shader to use. Before the fragment shaders run, clipping is performed. Clipping discards all fragments that are outside your view, increasing performance.

Fragment Shader

The main purpose of the fragment shader is to calculate the final color of a pixel and this is usually the stage where all the advanced OpenGL effects occur.



```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(0.8f, 1.0f, 0.8f, 1.0f);
}
```

Per-Sample Operations

After all the corresponding color values have been determined, the final object will then pass through one more stage that we call the alpha test and blending stage. This stage checks the corresponding depth (and stencil) value of the fragment and uses those to check if the resulting fragment is in front or behind other objects and should be discarded accordingly. The stage also checks for alpha values (alpha values define the opacity of an object) and blends the objects accordingly. So even if a pixel output color is calculated in the fragment shader, the final pixel color could still be something entirely different when rendering multiple triangles.

Transformation:

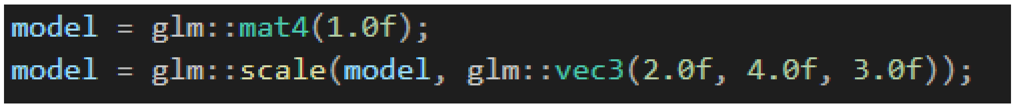
Moving the object by changing their vertices and re-configuring their buffers each frame we have to perform Transformation. In specific to our Project, we used vertex data of a Box and transformed multiple times for forming our Maze and World.

GLM

We used Open**GL** **M**athematics (GLM) library for performing different types of mathematical operations. OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications.

Scaling:

Scaling is increasing the length by certain amount (scaling factor) keeping the direction constant.

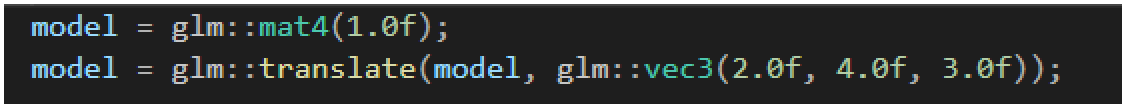


```
model = glm::mat4(1.0f);
model = glm::scale(model, glm::vec3(2.0f, 4.0f, 3.0f));
```

Here, we scaled the model matrix in x-axis by 2.0, y-axis by 4.0f and z-axis by 3.0f.

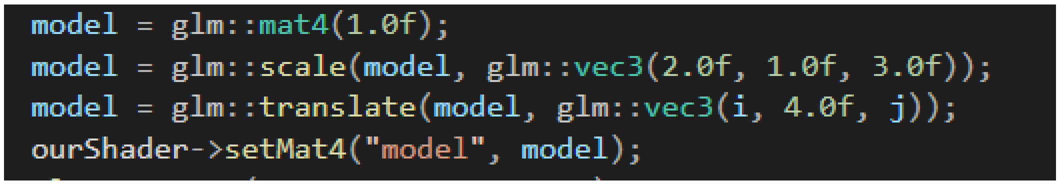
Translation:

Translation is the process of adding another vector on top of the original vector to return a new vector with a different position, thus moving the vector based on a translation vector.



```
model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(2.0f, 4.0f, 3.0f));
```

Combining these two-translation operations in matrix, the translation matrix is passed to vertex shader.



```
model = glm::mat4(1.0f);
model = glm::scale(model, glm::vec3(2.0f, 1.0f, 3.0f));
model = glm::translate(model, glm::vec3(i, 4.0f, j));
ourShader->setMat4("model", model);
```

Camera:

Camera acts like an eye for the game. In our First person moving game, camera plays the vital role for viewing the world from camera's perspective as origin of the scene . Transforming the world coordinates into viewing coordinates that are related to camera's position and direction. SO, camera defines the viewing space. To define the camera, we need camera position, Target direction, right axis and Up axis.

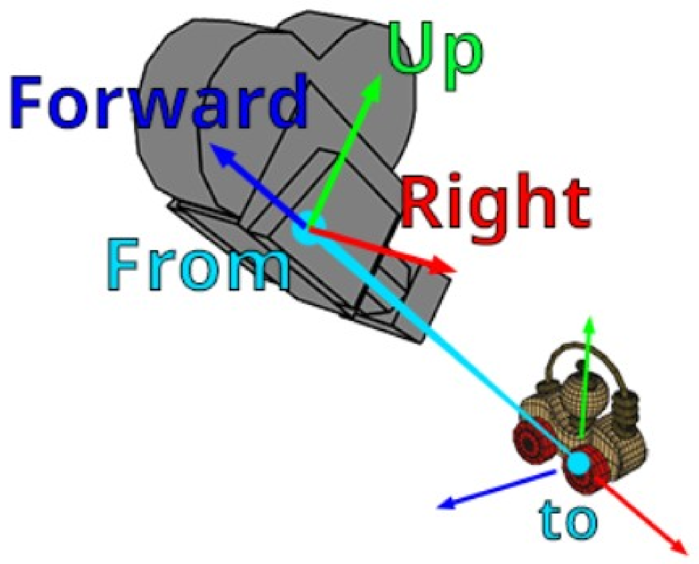


Figure 2 Local coordinate system of camera aimed at a point

Camera Position: In our First-person game, player position defines the camera position. Player position needs to be computed first which is controlled from keyboard's input. We use GLFW to track the keyboard inputs for certain keyboard pressed.

Camera direction: Direction vector of camera defines what direction the camera is looking at. Direction is aligned along the line segment defined by the points *from* and *to*.

```
Vec3f direction = Normalize (from - to);
```

Right axis: Right axis or Right vector defines the x-axis of the camera. While computing the right vector, we arbitrarily define some random vector (generally y-axis of World co-ordinate system) and cross product with right vector which gives us the vector perpendicular to both which is our right vector.

```
Vec3f tmp(0, 1, 0);
Vec3f right = crossProduct(Normalize(tmp), forward);
```

UP vector: We already have two orthogonal vectors (Direction and Right vector), so computing the cross product of two vector will just give us the Up vector.

```
Vec3f up = crossProduct(forward, right);
```

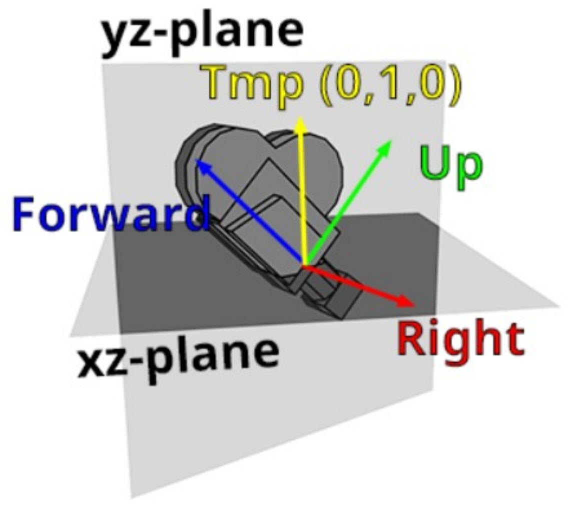


Figure 3 Computing axes of the camera

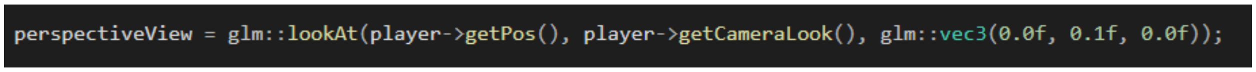
Look At: Three orthogonal coordinate space of camera defines the viewing space. Matrix representation of this viewing space need to be transformed in matrix form. Look AT matrix of the camera is passed to the shader as the viewing matrix.

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

Figure 4 Look At matrix

where R is Right vector, U is up vector and D is direction vector. The LookAt matrix does exactly what it says, it creates a view matrix that looks at the given target.

GLM already provide us the function to calculate the LookAt matrix.



```
perspectiveView = glm::lookAt(player->getPos(), player->getCameraLook(), glm::vec3(0.0f, 0.1f, 0.0f));
```

The glm: LookAt function requires a position, target and up vector respectively.

Projection:

Rendering the 3D- object in our 2D screen is quite cumbersome task. Our object formed by 3D- coordinate in camera space need to be projected on to the projection plane. This projection onto the plane is done the by the projection matrix. There are two types of projection i.e. orthographic projection and perspective projection. However, our game is first person viewing game so perspective projection is only used.

Perspective projection: Perspective projection produces the realistic result as the world will be rendered in perspective to the player eye/position. Perspective projection lets us enjoy graphics the real life has to offer giving the realistic view. The projection matrix maps a given frustum range to clip space.

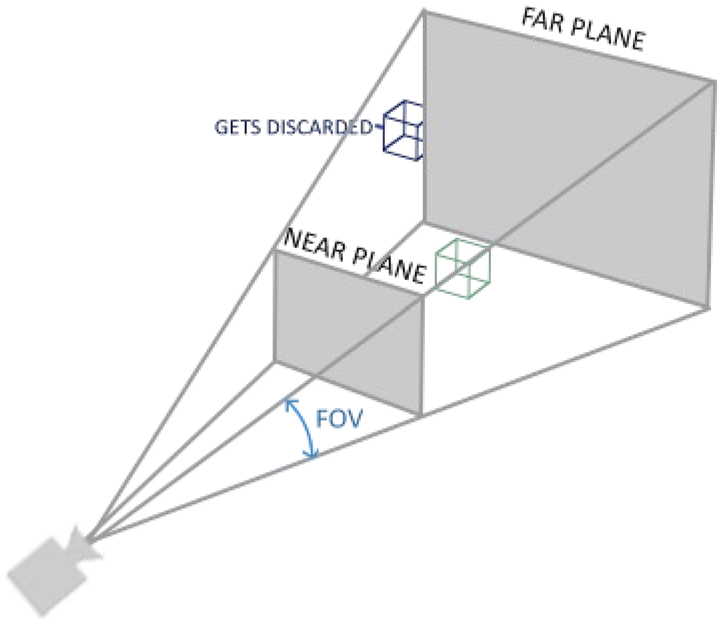
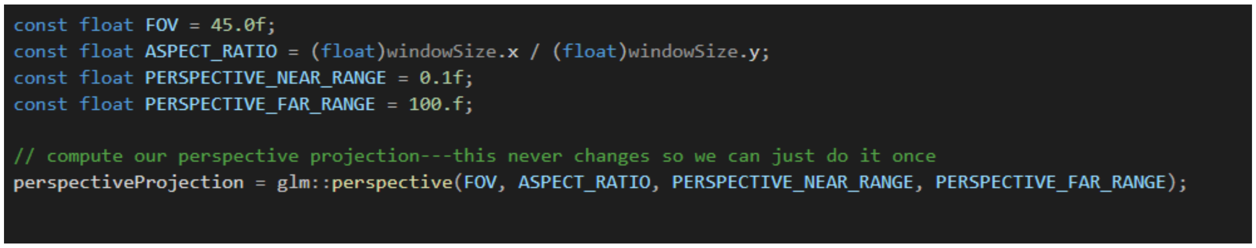


Figure 5 Perspective projection frustum

A perspective projection matrix can be created in GLM. *glm::perspective* create a large frustum that defines the visible space, anything outside the frustum will not end up in the clip space volume and will thus become clipped.



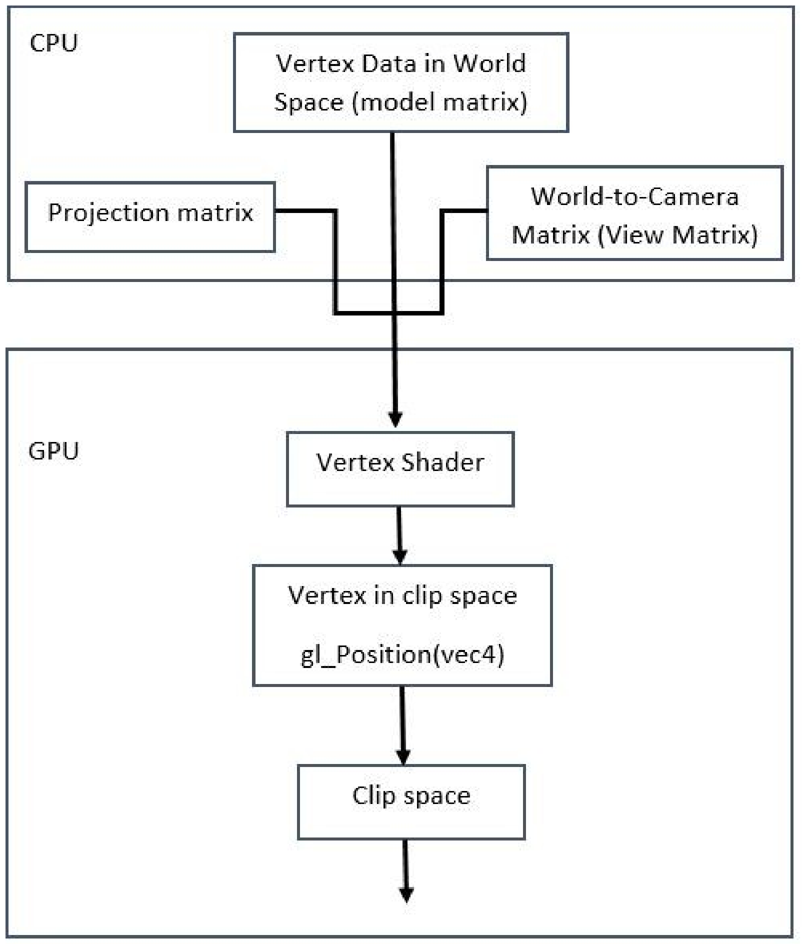
```
const float FOV = 45.0f;
const float ASPECT_RATIO = (float)windowSize.x / (float)windowSize.y;
const float PERSPECTIVE_NEAR_RANGE = 0.1f;
const float PERSPECTIVE_FAR_RANGE = 100.f;

// compute our perspective projection---this never changes so we can just do it once
perspectiveProjection = glm::perspective(FOV, ASPECT_RATIO, PERSPECTIVE_NEAR_RANGE, PERSPECTIVE_FAR_RANGE);
```

Rendering workflow:

Vertex Data stored in vertex buffers and used multiple times while rendering the scenes. Vertex buffer stores 3D- position, texture coordinates and normal which are transformed multiple times using the different transformation matrices. These vertex attributes modeled by the matrix, transformed by transformation matrix and sent to shader which is stored in GPU.

Projecting the objected visible in frustum is projection onto the projection plane which is done by the perspective view. Defining the far plane and near plane and using the viewport size as size of projection plane is then transferred to vertex shader using uniforms.

In the complete world, we can only view the certain portion using view matrix. We can rotate our camera taking mouse input and compute yaw, pitch and roll for calculating view matrix. We use GLFW to handle mouse event by registering the call back function. Thus computed view matrix is sent to shader.



Lighting

We implemented the phong lighting model for our project.

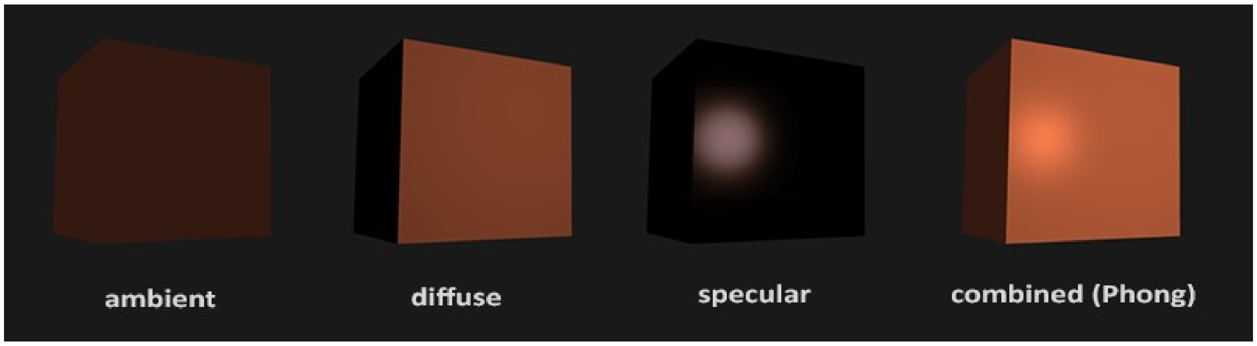


Figure 6: Components of Phong Lighting Model

The major building blocks of the Phong lighting model consist of 3 components: ambient, diffuse and specular lighting. Below you can see what these lighting components look like on their own and combined:

Ambient lighting:

Even when it is dark there is usually still some light somewhere in the world (the moon, a distant light) so objects are almost never completely dark. To simulate this, we use an ambient lighting constant that always gives the object some color.

```
void main()
{
float ambientStrength = 0.1;
vec3 ambient = ambientStrength *
lightColor;

vec3 result = ambient * objectColor;
FragColor = vec4(result, 1.0);
}
```

Figure 7: Ambient Lighting

Diffuse lighting:

Diffuse lighting simulates the directional impact a light object has on an object. This is the most visually significant component of the lighting model. The more a part of an object faces the light source, the brighter it becomes.

To the left we find a light source with a light ray targeted at a single fragment of our object. We need to measure at what angle the light ray touches the fragment.

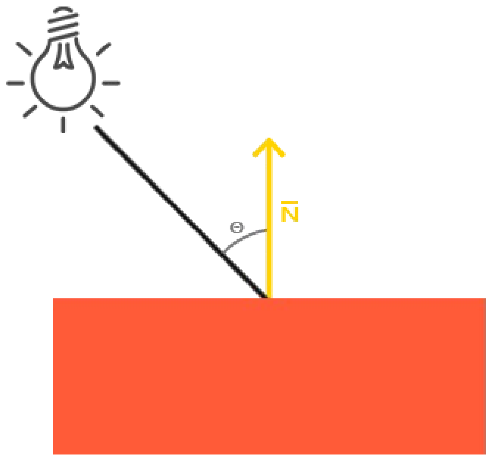


Figure 8: Diffused Lighting

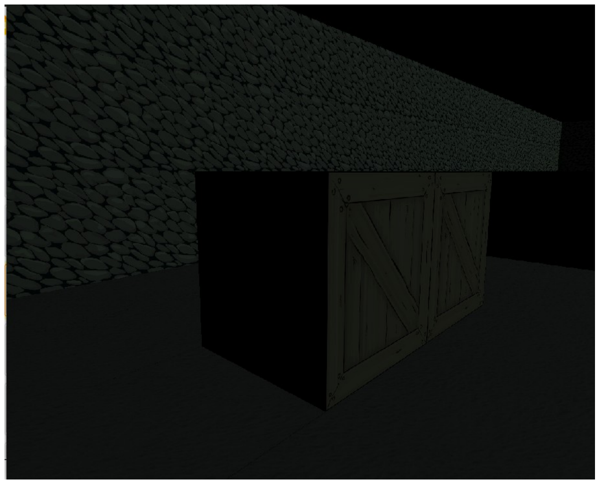


Figure 9: Effect of diffused lighting

If the light ray is perpendicular to the object's surface the light has the greatest impact. To measure the angle between the light ray and the fragment we use something called a normal vector, that is a vector perpendicular to the fragment's surface (here depicted as a yellow arrow); we'll get to that later. The angle between the two vectors can then easily be calculated with the dot product.

The resulting dot product thus returns a scalar that we can use to calculate the light's impact on the fragment's color, resulting in differently lit fragments based on their orientation towards the light.

```
vec3 norm = normalize(Normal);
vec3 lightDir = normalize(lightPos - FragPos);
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = diff * lightColor;
```

Specular lighting:

Specular lighting simulates the bright spot of a light that appears on shiny objects. Specular highlights are more inclined to the color of the light than the color of the object.

We calculate a reflection vector by reflecting the light direction around the normal vector. Then we calculate the angular distance between this reflection vector and the view direction. The closer the angle between them, the greater the impact of the specular light. The resulting effect is that we see a bit of a highlight when we're looking at the light's direction reflected via the surface.

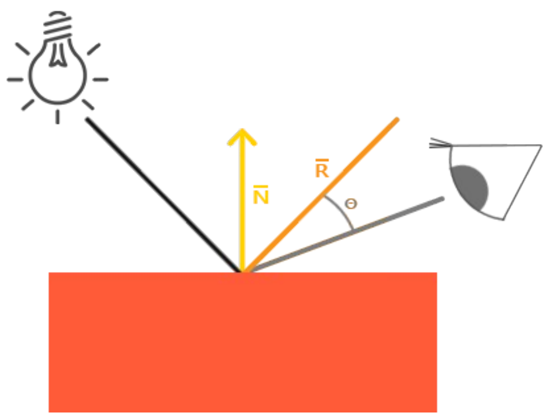


Figure 10 Specular Lighting

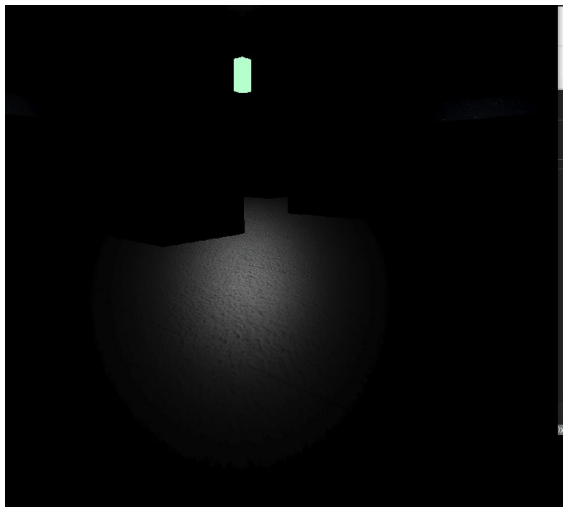


Figure 11: Effect of specular lighting

The view vector is the one extra variable we need for specular lighting which we can calculate using the viewer's world space position and the fragment's position. Then we calculate the specular's intensity, multiply this with the light color and add this to the ambient and diffuse components.

```
float specularStrength = 0.5;
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
vec3 specular = specularStrength * spec * lightColor;
```

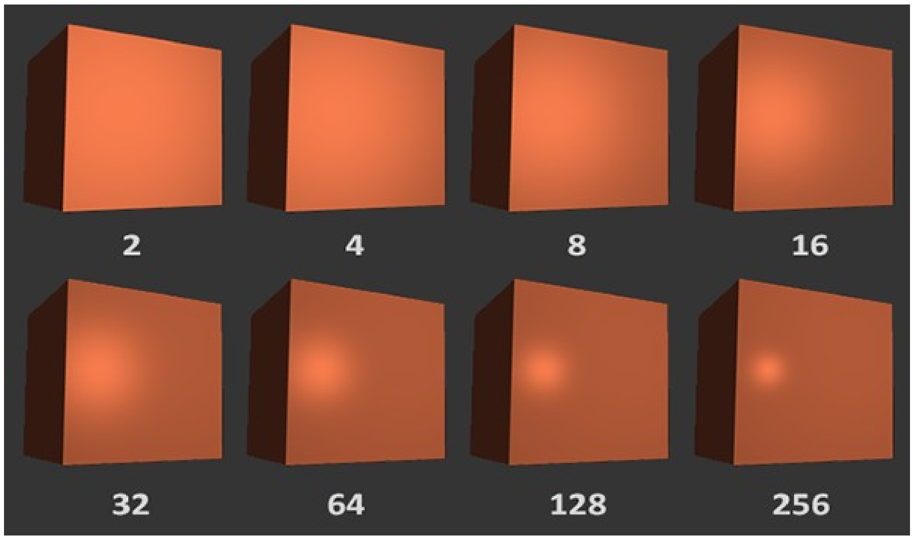


Figure 12: Result of specular lighting for different values of power term

```
vec3 result = (ambient + diffuse + specular) * objectColor;
FragColor = vec4(result, 1.0);
```

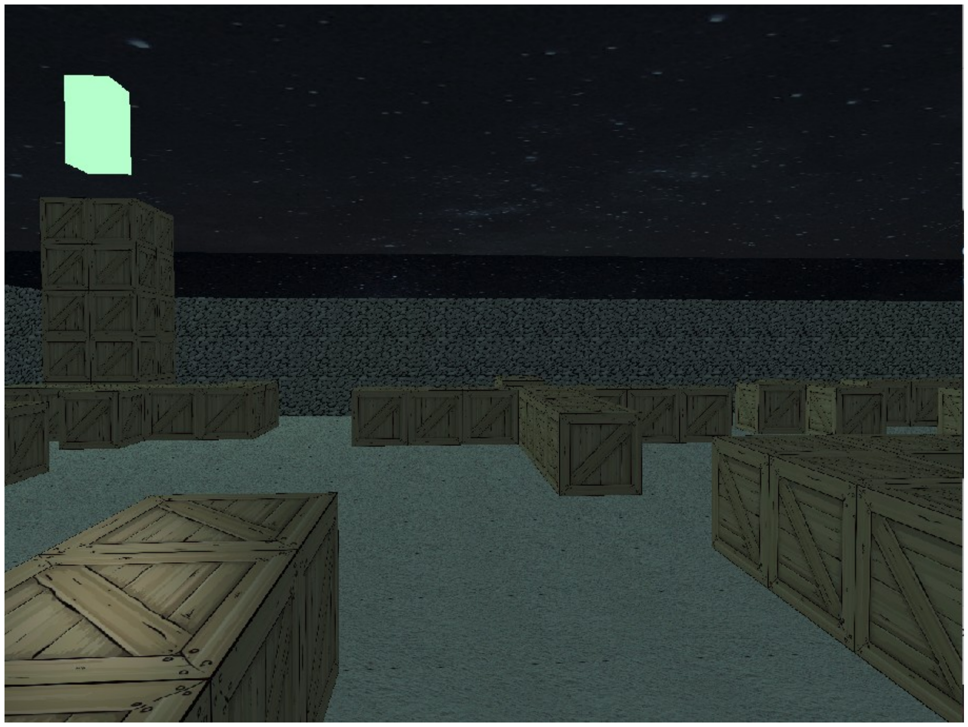


Figure 13: Final Result after adding all components of phong model

The ambient and diffusion strength were made low such that it would appear as if it is at night. The light color was also given a greenish hue.

Textures

Uniformly colored 3D objects look nice enough, but they are a little bland. Their uniform colors don't have the visual appeal of, say, a brick wall or a plaid couch. Three-dimensional objects can be made to look more interesting and more realistic by adding a texture to their surfaces. A texture, in general, is some sort of variation from pixel to pixel within a single primitive. We have used only one kind of texture: image textures. An image texture can be applied to a surface to make the color of the surface vary from point to point, something like painting a copy of the image onto the surface. Let's have a look at the cubes shown below:

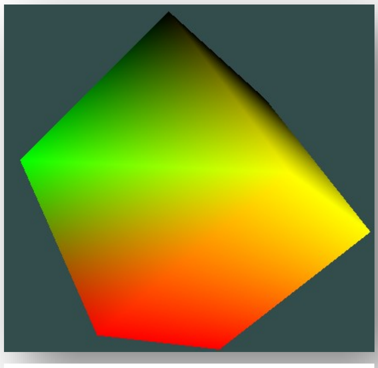


Figure 14: Colourful cube

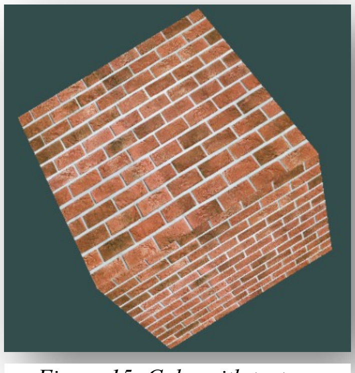


Figure 15: Cube with texture

The cube on the left looks colorful and definitely eye catching but it doesn't represent anything in real world scene. On the other hand, the cube on the right is not as colorful as the left one but it easy to say, it might be a piece of wall or a block of brick. Therefore, textures play a vital role in graphics to make objects looks realistic, either it be a 2D scene or a 3D scene. Whenever we talk about texture, we mean a 2D image not that a texture can't be 3D or even 1D but we are only using 2D texture for this project.

Texture Coordinates:

When a texture is applied to a surface, each point on the surface has to correspond to a point in the texture. There has to be a way to determine how this mapping is computed. For that, the object needs texture coordinates. As is generally the case in OpenGL, texture coordinates are specified for each vertex of a primitive. Texture coordinates for points inside the primitive are calculated by interpolating the values from the vertices of the primitive.

A texture image comes with its own 2D coordinate system. Traditionally, **s** used for the horizontal coordinate on the image and **t** is used for the vertical coordinate. The **s** coordinate is a real number that ranges from **0 on the left of the image to 1 on the right**, while **t ranges from 0 at the bottom to 1 at the top**. Values of **s** or **t** outside of the range 0 to 1 are not inside the image, but such values are still valid as texture coordinates. Note that texture coordinates are not based

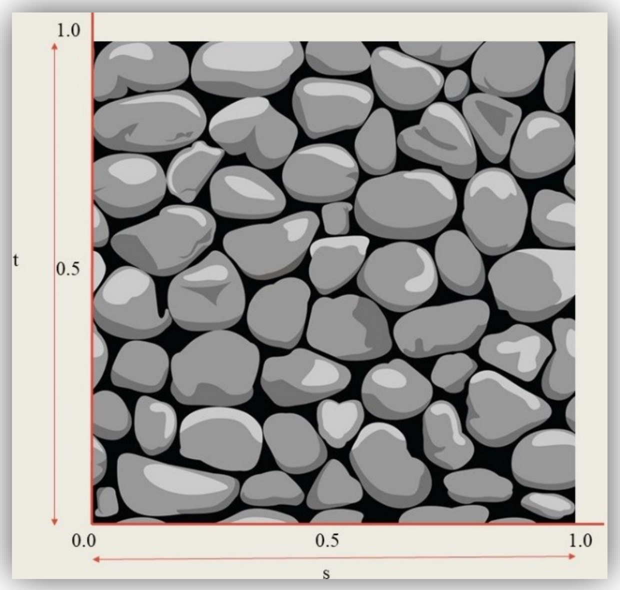


Figure 16: Texture coordinate2

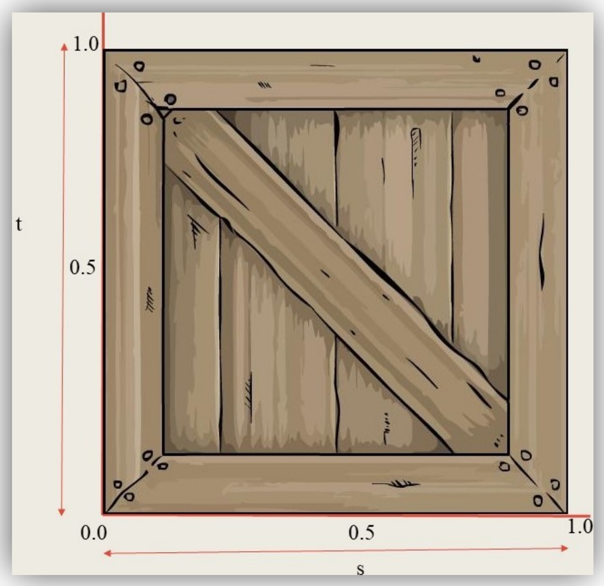
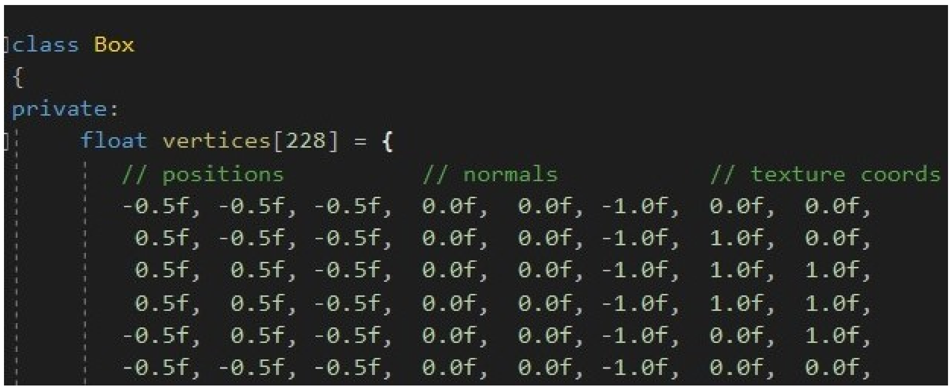


Figure 17: Texture coordinate

on pixels. No matter what size the image is, values of **s** and t between 0 and 1 cover the entire image.

To draw a textured primitive, we need a pair of numbers (s, t) for each vertex. These are the texture coordinates for that vertex. They tell which point in the image is mapped to the vertex.



```
class Box
{
private:
    float vertices[228] = {
        // positions          // normals           // texture coords
        -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  0.0f,
         0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  0.0f,
         0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  1.0f,
         0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  1.0f,
        -0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  1.0f,
        -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  0.0f,
```

Figure 18: Texture Coordinate, a vertex attribute

The texture coordinates of a vertex are an attribute of the vertex, just like color, normal vectors, and material properties.

Sampling and Wrapping

Retrieving the texture color using texture coordinates is called sampling. In order to get more detail in the object's color, a texture is used to pick color values from. We could define separate colors for each of the vertices and then interpolate between them to get a color for the fragment. Figure2 is a perfect example of a 3D object mapped to an image and color for each vertex is sampled from the image itself. The inside detail of sampling is beyond the scope of this project, so we won't be talking about it in detail. Fragment shader does the job for us.

Texture coordinates usually range from (0,0) to (1,1) but what happens if we specify coordinates outside this range? The default behavior of OpenGL is to repeat the texture images (the integer part of the floating-point texture coordinate is ignored), but there are more options OpenGL. Out of all options GL_REPEAT served our purpose well. The behaviour of this option is self-explanatory.

Filtering and Mipmap

When a texture is applied to a surface, the pixels in the texture do not usually match up one-to-one with pixels on the surface, and in general, the texture must be stretched or shrunk as it is being mapped onto the surface. Sometimes, several pixels in the texture will be mapped to the same pixel on the surface. In this case, the color that is applied to the surface pixel must somehow be computed from the colors of all the texture pixels that map to it. This is an example of "filtering"; in particular, it uses a minification filter because the texture is being shrunk. When one pixel from the texture covers more than one pixel on the surface, the texture has to be magnified, and we need a magnification filter. The pixels in a texture are referred to as texels, short for "texture pixel" or "texture element".

When deciding how to apply a texture to a point on a surface, OpenGL knows the texture coordinates for that point. Those texture coordinates correspond to one point in the texture, and that point lies in one of the texture's texels. The easiest thing to do is to apply the color of that texel to the point on the surface. This is called "nearest texel filtering." It is very fast, but it does not usually give good results. It doesn't take into account the difference in size between the pixels on the surface and the texels in the image. An improvement on nearest texel filtering is "linear filtering," which can take an average of several texel colors to compute the color that will be applied to the surface.

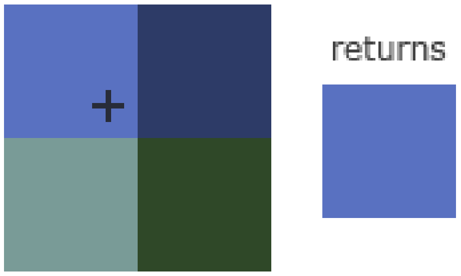


Figure 19: GL_NEAREST

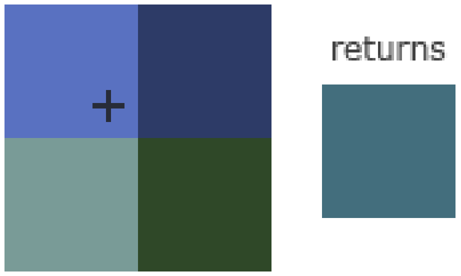
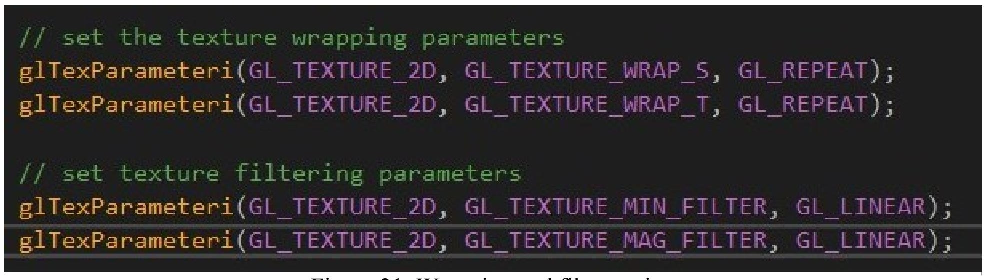


Figure 20: GL_LINEAR



```
// set the texture wrapping parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

// set texture filtering parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Figure 21: Wrapping and filter setting

The problem with linear filtering is that it will be very inefficient when a large texture is applied to a much smaller surface area. In this case, many texels map to one pixel, and computing the average of so many texels becomes very inefficient. There is a neat solution for this: mipmaps.

A mipmap for a texture is a scaled-down version of that texture. A complete set of mipmaps consists of the full-size texture, a half-size version in which each dimension is divided by two, a quarter-sized version, a one-eighth-

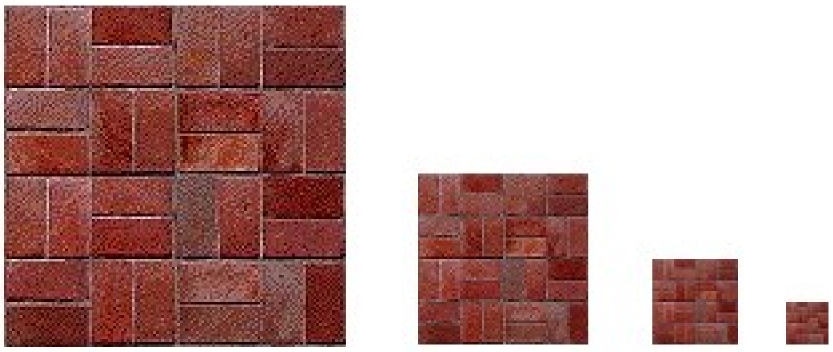


Figure 22: Mipmap

sized version, and so on. If one dimension shrinks to a single pixel, it is not reduced further, but the other dimension will continue to be cut in half until it too reaches one pixel.

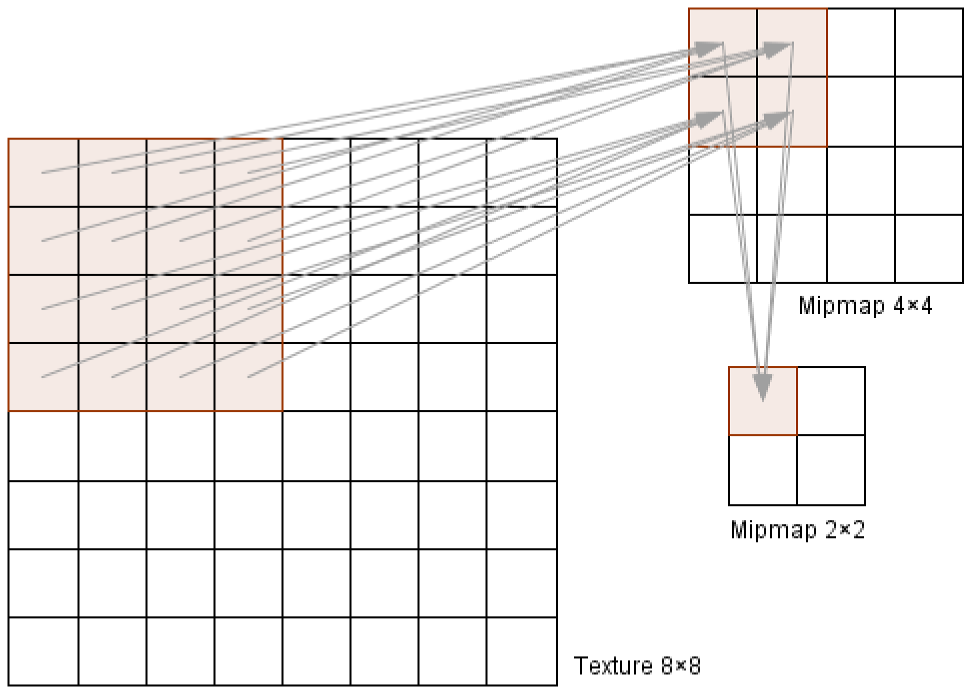
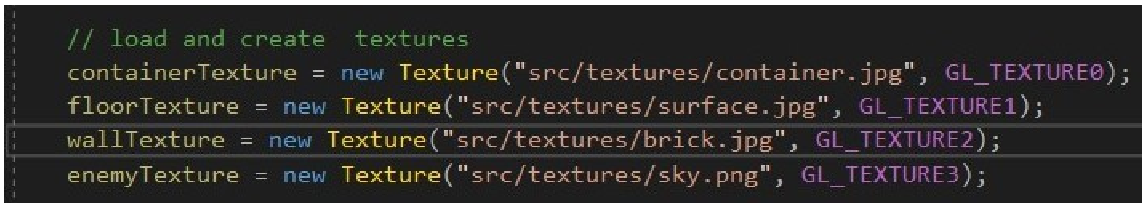


Figure 23: Mipmap process

Texture Units

The sampler in fragment shader must be bound to a texture unit. Texture units are references to texture objects that can be sampled in a shader. Textures are bound to texture units using the glBindTexture (). The main purpose of texture units is to allow us to use more than 1 texture in our shaders. By assigning texture units to the samplers, we can bind to multiple textures at once as long as we activate the corresponding texture unit first. The number of texture



```
// load and create  textures
containerTexture = new Texture("src/textures/container.jpg", GL_TEXTURE0);
floorTexture = new Texture("src/textures/surface.jpg", GL_TEXTURE1);
wallTexture = new Texture("src/textures/brick.jpg", GL_TEXTURE2);
enemyTexture = new Texture("src/textures/sky.png", GL_TEXTURE3);
```

units supported by a system depends in its graphics card.

Figure 24: Texture Units

Algorithm

1. Visible Surface Detection (Z-Buffer or Depth-Buffer method)

When viewing a picture containing non transparent objects and surfaces, it is not possible to see those objects from view which are behind from the objects closer to eye. To get the realistic screen image, removal of these hidden surfaces is must. The identification and removal of these surfaces is called as the **Hidden-surface problem.**

Z-buffer, which is also known as the Depth-buffer method is one of the commonly used method for hidden surface detection. It is an **Image space method**. Image space methods are based on the pixel to be drawn on 2D. For these methods, the running time complexity is the number of pixels times number of objects. And the space complexity is two times the number of pixels because two arrays of pixels are required, one for frame buffer and the other for the depth buffer.

The Z-buffer method compares surface depths at each pixel position on the projection plane. Normally z-axis is represented as the depth.

2. Obstacle Detection

Algorithm to find obstacles around a player:

```
1. Read playerPos = vec3 ($x_1$, $y_1$, $z_1$)

2. positionOnXZplane = vec3 (int ($x_1$ + 0.5), 0, int ($z_1$ + 0.5))

3. for p = -1 to 1
       for q = -1 to 1
           IF (map[positionOnXZplane.x + p][positionOnXZplane.z + q] is an obstacle)
               locations.push(positionOnXZplane.x + p, positionOnXZplane.z + q)
   return locations
```

Test collision with an obstacle

```
1. Read enemyPos = vec3 ($x_1$, $y_1$, $z_1$)

2. Read testingLocation = ($x_2$, $z_2$)
```

```
3. If( enemyPos.x <= testingLocation.x + 0.65 and enemyPos.x >= testingLocation.x – 0.65)

          If(enemyPos.z <= testingLocation.z + 0.65 and enemyPos.z >= testingLocation.z –
   0.65

          return true

   return false
```

Movement Restriction

```
glm::vec3 vecfromcamtoobstacle = glm::normalize(glm::vec3(location.x, 0.0f, location.z)
- *cameraPos);

            if (fabs(vecfromcamtoobstacle.x) > fabs(vecfromcamtoobstacle.z))

                  cameraPos->x = oldCameraPos.x;

            else if (fabs(vecfromcamtoobstacle.x) < fabs(vecfromcamtoobstacle.z))

            {

                  cameraPos->z = oldCameraPos.z;

            }

            else

            {

                  cameraPos->x = oldCameraPos.x;

                  cameraPos->z = oldCameraPos.z;

            }
```

3. Algorithm for enemy's vision

- Read enemyPos = vec3 (x_1, y_1, z_1)
- Read playerPos = vec3 (x_2, y_2, z_2)
- Set parameter = 0.5
- Calculate the direction vector from enemyPos to playerPos

```
            enemyToplayer = normalize (playerPos – enemyPos)
▪   enemytoPlayerDistance = length (playerPos – enemyPos)
▪   newEnemyPos = enemyPos + (parameter * enemyToplayer)
▪   while (parameter < enemytoPlayerDistance)
            locations = findObstaclesTowardPlayer (newEnemyPosition)
            for each location: locations
            if (isColliding (newEnemyPos, location))
                    return false
            parameter += 0.5
            newEnemyPos = enemyPos + (parameter * enemyToplayer)
            if (parameter > enemytoPlayerDistance)
            return true
    return false
```

4. A* implementation

```
   i. OPEN = priority queue containing START
  ii. CLOSED = empty set
 iii. while lowest rank in OPEN is not the GOAL:
  iv. current = remove lowest rank item from OPEN
   v. add current to CLOSED
  vi. for neighbors of current:
 vii. cost = g(current) + movementcost (current, neighbor)
viii. if neighbor in OPEN and cost less than g(neighbor):
  ix. remove neighbor from OPEN, because new path is better
```

```
    x. if neighbor in CLOSED and cost less than g(neighbor): (2)
   xi. remove neighbor from CLOSED
  xii. if neighbor not in OPEN and neighbor not in CLOSED:
 xiii. set g(neighbor) to cost
  xiv. add neighbor to OPEN
   xv. set priority queue rank to g(neighbor) + h(neighbor)
  xvi. set neighbor's parent to current
 xvii. reconstruct reverse path from goal to start
xviii. by following parent pointers
```

Game play

Once the scene is rendered on the screen, a player and an enemy always start at their respective position. The player starts at vec3(1.0, 0.5, 1.0) while the enemy starts at vec3(1.0, 0.0, 15.0). The job of the player is to deceive the enemy and make it to the tower situated at the opposite corner. He can hide himself behind the obstacles lying around and find a way out to the target but, the enemy is constantly looking for the player. As soon as the enemy catches the player in his sight, an alarm goes off. It is not just the alarm but the big problem is, now, the enemy can travel much quickly than before. It is player’s job to find a safe position to hide before the enemy catches him. When the player goes out of sight, searching begins for the enemy which will buy some more time for the player to plan his steps ahead.

What if the player is caught? Well, all his hard work goes down to the gutter and he is sent back to the starting position. Good news is, even the enemy must go back to his original position and start all over again. The game is over only when the player makes to the tower safely. Second round of play is always open to start the journey again.

Player Movement:

Player can move around using A, S, W and D keys and change his field of view using mouse.

Movement	**Key(s)**
Forward	W
Backward	S
Right	D
Left	A

Enemy Movement

A* implementation creates a shortest path from enemy object to the player object. The enemy will then move on to the next node in the list of the path provided by the algorithm. This way the enemy object will keep on following the player. The determined path can change in runtime and hence whenever the player moves to new position the enemy can determine and follow along the shortest path to the player in runtime.

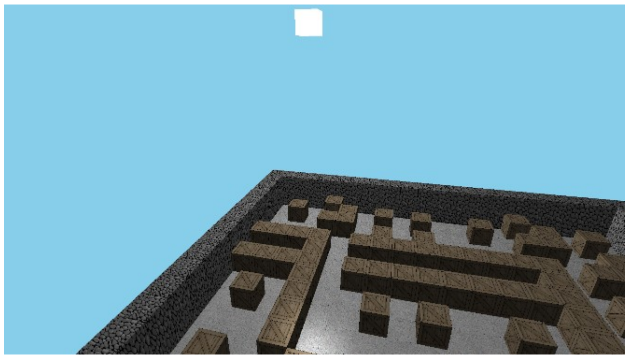


Figure 25: Rendered Map

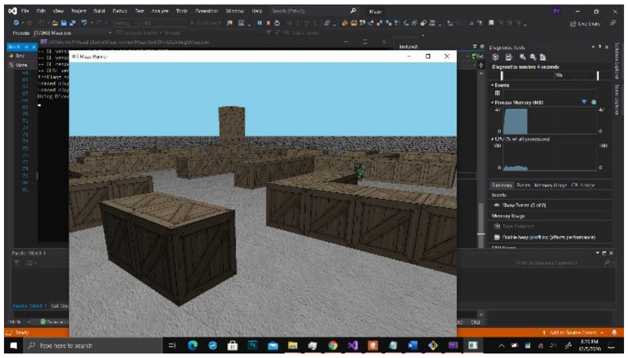


Figure 26 : Player's view

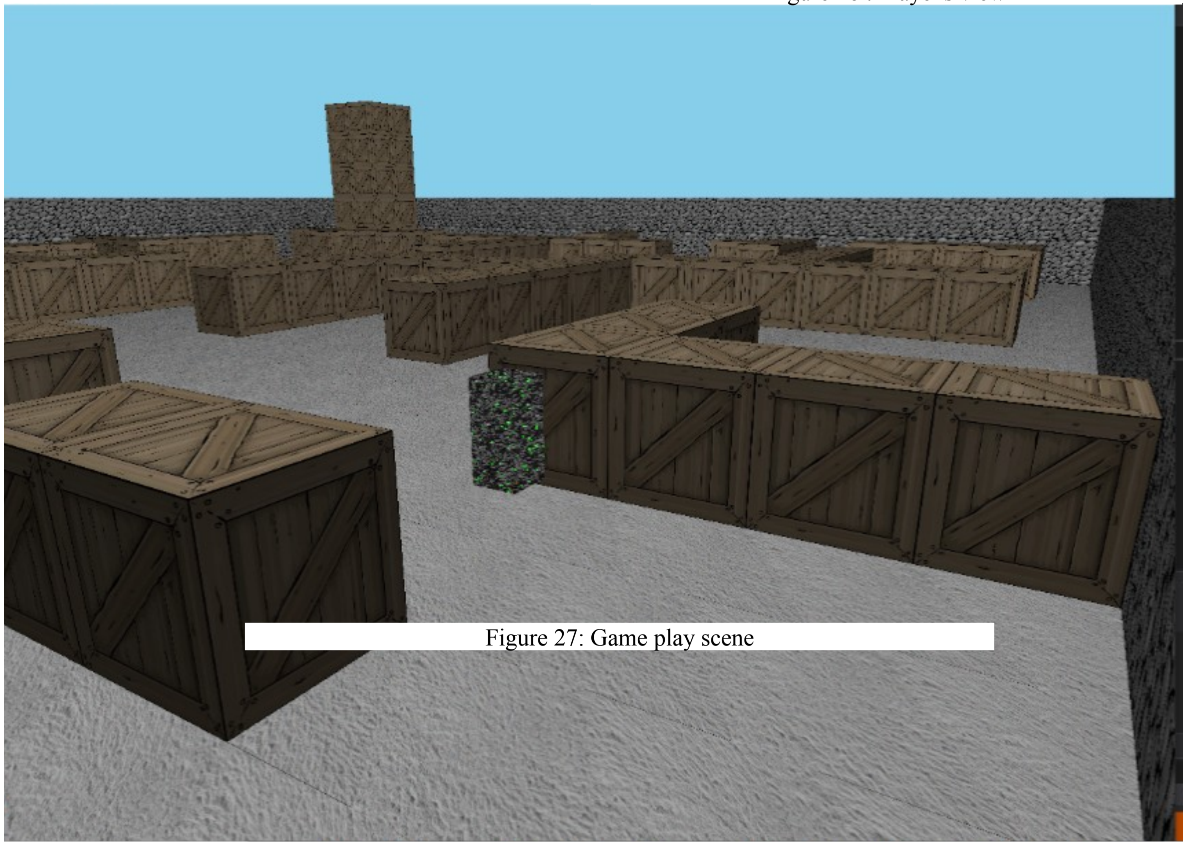


Figure 27: Game play scene

Final Analysis

Result

As we had intended at the beginning of this project to create a 3D game rendering engine using OpenGL 3.3 and GLSL 330, we have achieved our goal. The scene looks close to real if not realistic, different elements of computer graphics as discussed before have been used to make it possible. A* implementation for path finding was next big task. In the end, we have successfully implemented it. With limited knowledge of OpenGl and computer graphics in whole, we are proud to have our goal accomplished.

Here, we have the screenshots of the application in execution where the enemy is following the player in the genereted map.

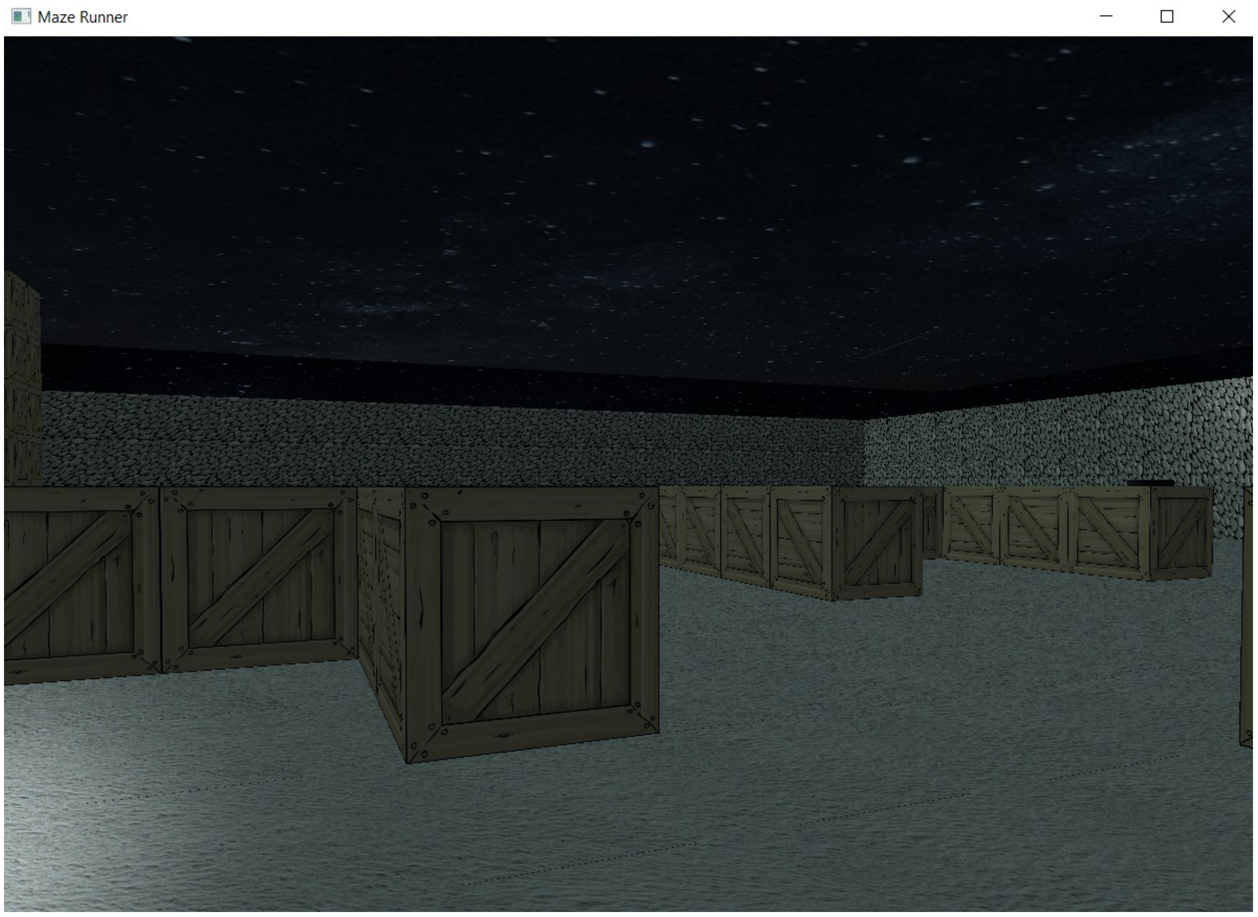


Figure 28: Final Game Scene

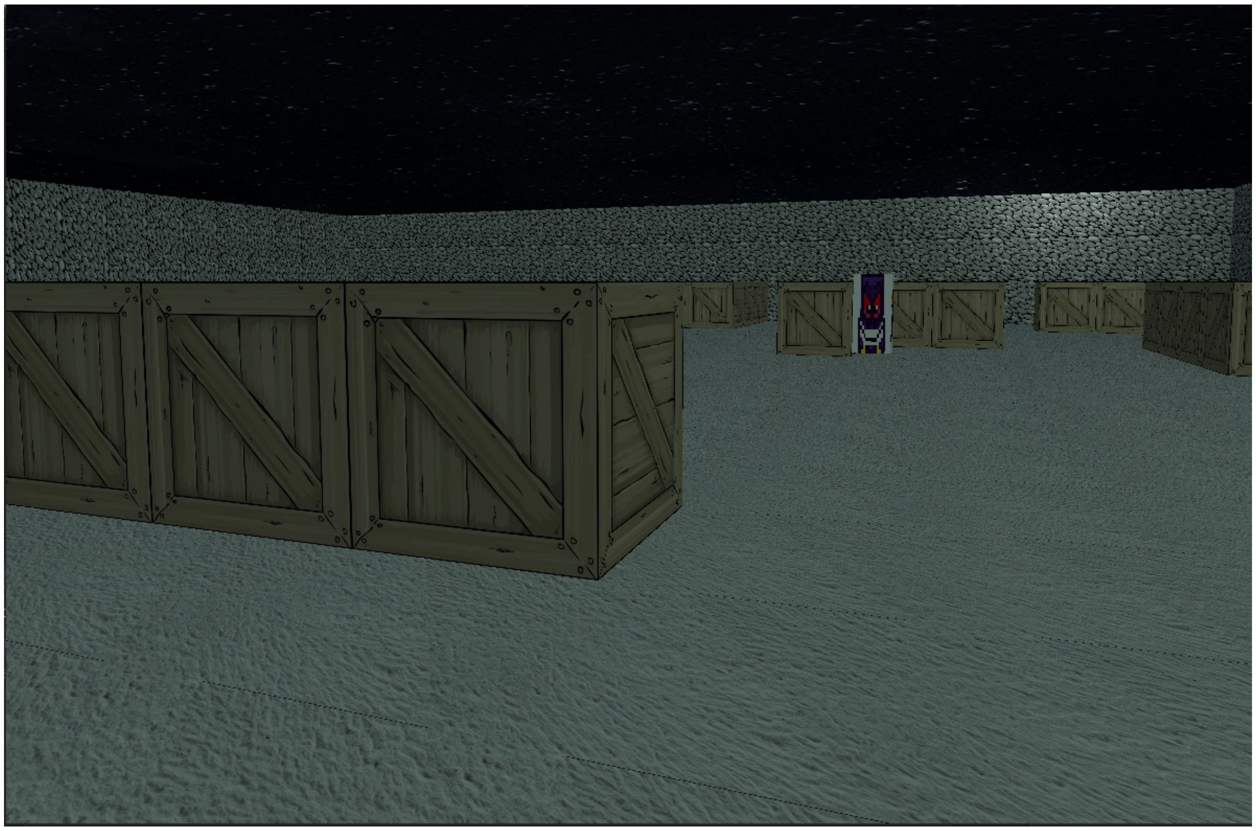


Figure 29: Enemy chasing the player

Application

The game engine is rendered using OpenGL pipeline taking inspiration from FPS games like Doom and Wolfenstein 3D. It can further be upgraded to make something similar to such games. The vital part of the game play is pathfinding which depends on implementation of A* algorithm. We can use A* code as implementing code which can be reused, implementing in other application. This program gives the general idea for pathfinding. Algorithms like these are generally used in game engine for NPC (Non-Player Character) in order to find the path as well as best possible states for animation and to make fluid actions via decision making.

Limitation

Rendering Approach: Right now, we are drawing the entire scene in every frame, which means hundreds of draw calls in very frame. This approach takes a lot of resources and it is not a good way to render a 3D scene.

Program limitation: In this project, we calculate the whole shortest path for determining the direction the enemy should be moving in. This is wasted resources and time which can lead to lag when the determined path is long.

Application limitation: Due to the limitation mentioned above the application may render less frames when the resources are being used by A* algorithm to determine longer routes.

With improved 3D rendering approach like Binary space partition, its performance can be vastly improved.

Problem Faced

Main problem faced during the development of the program is how we can work independently and how and from where to start. Question raises at every instance of coding as we all are new to OpenGL, Git and algorithm implementation. Using Git for collaboration and working on program was also new. Also optimizing the code for the application to run smoothly and render sufficient frames for the game to feel smooth and not be very resource hungry was one of the major challenges.

Conclusion

The game engine is performing very well in our machine. However, it is not possible to say it will do so even in low spec computer based on our tests. All of us have fairly good graphics card and latest CPUs. With improvement in rendering technique, we are confident that it will perform fairly well in low spec machines too.

A* algorithm performs exceptionally well in terms of finding the most efficient path in a reasonable time. In this project, the fps remained reasonably high even when A* was called in each render loop. Thus, it is very suitable for path planning in games. The collision detection algorithm performed reasonably well with few glitches. The path planning can be further optimized by using a previously calculated path for more than a single render loop. The no. of iterations for which a single calculated path can be used can be determined using a heuristic based on how fast the player and the enemy move along with how close they are. The 2D collision detection can be expanded along another dimension to allow jumping over obstacles and can be made more accurate by using a bounding box for the player instead of just a point in space.

Source Code

Main.cpp

```
//#define DEBUG
#include "GL/glew.h"
#include "world.h"
#include "GLFW/glfw3.h"

#include "glm/glm.hpp"
#include <ctime>
#include<cstdlib>
#include <iostream>


// GLFW window and characteristics
GLFWwindow* window;
static glm::vec2 windowSize;

void openWindow();
void prepareOpenGL();

int main(void)
{
	// some important objects
	World* world;

	// used to control frame timing
	const double TARGET_FRAME_INTERVAL = 0.016665;		// slightly smaller than
1/60, our desired time (sec.) between frames
	const double FRAME_TIME_ALPHA = 0.25;				// degree to
which previous frame time is used for delta time computation

	// these handle our very smooth frame timing
	double dt;
	// time since last cycle through loop (*not* time between update/render steps)
	double currentTime;							//
current time according to GLFW
	double oldTime;
	// what the value of currentTime was on the last cycle through loop
	double frameTime = 0.0;							// delta
time of the next frame to render to keep things smooth
	double renderAccum = 0.0;						// accumulated
time between update/render steps
	double smoothFrameTime = TARGET_FRAME_INTERVAL;		// weighted
average delta time used to smooth out next frame
```

```
	// reseed with the clock
	srand(unsigned(time(NULL)));

	// create our OpenGL window and context, and set some rendering options
	openWindow();

	//set opengl states
	prepareOpenGL();

	// construct our world using a PNG image that describes how it is built
	world = new World(window, windowSize);

	// prime our time tracking
	currentTime = glfwGetTime();
	oldTime = currentTime;

	// loop until we're done
	while (!glfwWindowShouldClose(window) && !glfwGetKey(window, 
GLFW_KEY_ESCAPE))
	{


		// compute a time multiplier based on our frame rate
		oldTime = currentTime;
		currentTime = glfwGetTime();
		dt = currentTime - oldTime;

		// track the time that has passed since the last loop cycle
		renderAccum += dt;// accumulated time between update/render steps
		frameTime += dt;  // delta time of the next frame to render to keep things smooth /

		// is it time to update the scene and render a new frame?
		if (renderAccum >= TARGET_FRAME_INTERVAL)
		{
			// clear our color and depth buffers
			glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

			// reset the timer for our next frame after this one
			renderAccum -= TARGET_FRAME_INTERVAL;

			// use our previous frame time to compute a smoother frame time for this 
update cycle
			smoothFrameTime = (smoothFrameTime * FRAME_TIME_ALPHA) + 
(frameTime * (1.0 - FRAME_TIME_ALPHA));
			frameTime = 0.0;
```

```
				// update the world state and render everything
				world->update(float(smoothFrameTime));
				world->render();

				// get input events and update our framebuffer
				glfwPollEvents();
				glfwSwapBuffers(window);
			}
		}

		delete world;

		// shut down GLFW
		glfwDestroyWindow(window);
		glfwTerminate();

		return 0;
}


void openWindow()
{
		const char* TITLE = "Maze Runner";

		int windowWidth = 1200;
		int windowHeight = 900;
		GLenum error;

		// we need to initialize GLFW before we create a GLFW window
		if (!glfwInit())
		{
			std::cerr << "openWindow() could not initialize GLFW" << std::endl;
			exit(1);
		}

		// explicitly set our OpenGL context to something that doesn't support any old-school shit
		glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
		glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
		glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
		glfwWindowHint(GLFW_REFRESH_RATE, 60);

#ifdef DEBUG
		glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GL_TRUE);
#endif

		/*
```

```
	// use the current desktop mode to decide on a suitable resolution
	const GLFWvidmode* mode = glfwGetVideoMode(glfwGetPrimaryMonitor());
	windowWidth = mode->width;
	windowHeight = mode->height;
	*/

	// create our OpenGL window using GLFW
	window = glfwCreateWindow(windowWidth, windowHeight,			// specify
width and height
		TITLE,									// title of window
//#ifdef DEBUG
		NULL,								// windowed mode
//#else
		//glfwGetPrimaryMonitor(),		// full screen mode
//#endif
		NULL);									// not sharing
resources across monitors
	glfwMakeContextCurrent(window);
	glfwSwapInterval(1);

	// disable the cursor
	glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

	// configure our viewing area
	glViewport(0, 0, windowWidth, windowHeight);

	// enable our extensions handler
	glewExperimental = true;				// GLEW bug: glewInit() doesn't get all
extensions, so we have it explicitly search for everything it can find
	error = glewInit();
	if (error != GLEW_OK)
	{
		std::cerr << glewGetErrorString(error) << std::endl;
		exit(1);
	}

	// clear the OpenGL error code that results from initializing GLEW
	glGetError();

	// save our window size for later on
	windowSize = glm::vec2(windowWidth, windowHeight);

	// print our OpenGL version info
	std::cout << "-- GL version:   " << (char*)glGetString(GL_VERSION) <<std::endl;
	std::cout << "-- GL vendor:    " << (char*)glGetString(GL_VENDOR) <<std::endl;
	std::cout << "-- GL renderer:  " << (char*)glGetString(GL_RENDERER) <<std::endl;
	std::cout << "-- GLSL version: " <<
(char*)glGetString(GL_SHADING_LANGUAGE_VERSION)<<std::endl;


}

void prepareOpenGL()
{
	// turn on depth testing and enable blending
	glEnable(GL_DEPTH_TEST);
	glEnable(GL_BLEND);
	glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

	// sky blue-ish in background
	glClearColor(0.0f, 0.0f, 0.0f, 0.0);
}
```

World.h

```
#pragma once
#include "glm/glm.hpp"
#include "Box.h"
#include "Player.h"



//class GLFWwindow;
class Player;
class Box;
class Enemy;


class World
{
private:
	GLFWwindow* window;
	Box* box;
	Player* player;
	Enemy* eVedar;

	glm::mat4 perspectiveProjection;					// 4x4 mat
describing a perspective projection (for the player's view)
	glm::mat4 perspectiveView;						// 4x4
mat describing how the perspective camera is oriented
	glm::mat4 orthoProjection;						// 4x4
mat describing an orthographic projection (for the HUD)
```

```
	glm::mat4 orthoView;								// 4x4
mat describing how the orthographic camera is oriented

	void controlCamera();

	// set up our camera matrices
	void preparePerspectiveCamera(glm::vec2 windowSize);
	void prepareOrthoCamera(glm::vec2 windowSize);



public:

	World(GLFWwindow* window, glm::vec2& windowSize);
	~World();

	// main updating and rendering
	void update(float dt);
	void render();
};
```

Player.h

```
#pragma once
#include "GL/glew.h"
#include "GLFW/glfw3.h"
#include "glm/glm.hpp"

#include "world.h"
#include "Enemy.h"
#include "Map.h"
#include "Collision.h"


//for soundEngine
#define IRRKLANG_STATIC
#include<irrKlang.h>

//class GLFWwindow;
class World;


class Player
{
private:
	GLFWwindow* window;
	World* world;//player needs to know the elements of the rendered world.

	Enemy* eVedar;  //for good guys to act they need to know

	irrklang::ISoundEngine* engine = NULL;
	irrklang::ISoundSource* soundFile = NULL;
	irrklang::ISoundSource* walkSound = NULL;

	glm::vec3 cameraPos ;
	glm::vec3 cameraFront;
	glm::vec3 cameraUp;

	// yaw is initialized to -90.0 degrees since a yaw of 0.0 results in a
	//direction vector pointing to the right so we initially rotate a bit to the left.
	float yaw ;
	float pitch ;
	float lastX ;
	float lastY ;
	float fov ;

	void controlMouseInput(float dt);		// turns mouse motion into camera angles

public:


	Player(GLFWwindow* window, World* world, Enemy* Vedar, glm::vec3 pos);
	~Player();

	//update routines
	void update(float dt);
	void processInput(float dt);

	glm::vec3 getPos();
	glm::vec3 getCameraUp();
	glm::vec3 getCameraLook();
	bool endlocation();
	bool EnemyReached();
	void reset();
};
```

Enemy.h

```
#pragma once
#ifndef ENEMY_H
#define ENEMY_H
#endif
```

```
#include <glm/glm.hpp>
#include "astar.h"
class Enemy
{
private:
	glm::vec3 myPosition;
	glm::vec3 directionToPlayer;

public:
	Enemy(glm::vec3& position);

	glm::vec3 getPosition();
	void updatePosition(glm::vec3& newPosition,float enemySpeed);
};
```

Box.h

```
#pragma once

#include "Shader.h"
#include "Player.h"
#include "texture.h"
#include "Enemy.h"
#include <iostream>

class Player;

class Box
{
private:
	 float vertices[288] = {
	// positions          // normals           // texture coords
	-0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  0.0f,
	 0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  0.0f,
	 0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  1.0f,
	 0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  1.0f,  1.0f,
	-0.5f,  0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  1.0f,
	-0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f,  0.0f,

	// …
	};
	Shader* ourShader;

	Player* player;
	Enemy* eVedar;
	Shader* lightCubeShader;
```

```
        Texture* containerTexture;
        Texture* floorTexture;
        Texture* wallTexture;
        Texture* enemyTexture;
        Texture* skyTexture;

        unsigned int VBO, VAO, lightCubeVAO;

public:
        Box(Player* player, Enemy* Vedar);
        ~Box();

        void render(glm::mat4& projection, glm::mat4& view);

};
```

Map.h

```
#pragma once
        const int MAP_HEIGHT = 25;
        const int MAP_WIDTH = 25;

        static int map[MAP_WIDTH][MAP_HEIGHT] = {
                {2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2},
                {2,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
                {2,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,1,1,0,0,2},
                {2,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,2},
                {2,0,0,1,0,0,1,1,1,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,2},
                {2,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,1,0,0,2},
                {2,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,2},
                {2,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,1,0,0,2},
                {2,1,1,1,1,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,1,0,0,0,2},
                {2,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,2},
                {2,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,2},
                {2,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,2},
                {2,0,0,0,1,0,0,0,0,1,1,1,1,0,0,0,0,0,0,1,0,0,0,0,2},
                {2,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,1,1,2},
                {2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,2},
                {2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,2},
                {2,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,2},
                {2,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
                {2,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2},
                {2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,2},
                {2,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,2},
                {2,0,0,1,0,0,0,0,1,1,1,1,0,1,1,1,1,0,0,1,1,0,0,0,2},
                {2,0,0,1,1,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,3,3,2},
                {2,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,3,3,2},
                {2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2}

        };
```

Shader.h

```
#pragma once
#ifndef SHADER_H
#define SHADER_H
#include <GL/glew.h>
#include "glm/glm.hpp"

#include <string>
#include <fstream>
#include <sstream>
#include <iostream>

class Shader
{
public:
   unsigned int ID;
   // constructor generates the shader on the fly
   Shader(const char* vertexPath, const char* fragmentPath, const char* geometryPath =
nullptr);

   // activate the shader
   void use();

   // utility uniform functions
   void setBool(const std::string& name, bool value) const;

   void setInt(const std::string& name, int value) const;

   void setFloat(const std::string& name, float value) const;

   void setVec2(const std::string& name, const glm::vec2& value) const;
   void setVec2(const std::string& name, float x, float y) const;

   void setVec3(const std::string& name, const glm::vec3& value) const;
   void setVec3(const std::string& name, float x, float y, float z) const;

   void setVec4(const std::string& name, const glm::vec4& value) const;
   void setVec4(const std::string& name, float x, float y, float z, float w);

   void setMat2(const std::string& name, const glm::mat2& mat) const;

   void setMat3(const std::string& name, const glm::mat3& mat) const;
```

```
    void setMat4(const std::string& name, const glm::mat4& mat) const;

private:
    // utility function for checking shader compilation/linking errors.
    void checkCompileErrors(GLuint shader, std::string type);
};
#endif
```

Texture.h

```
#pragma once
#ifndef TEXTURE_H
#define TEXTURE_H
#endif

#include <GL/glew.h>
#include "Third_Parties/stb_image.h"
#include <iostream>
class Texture
{
public:
    unsigned int ID;
    //unsigned char* data;

    //parameters: image path and texture unit
    Texture(const char* imagePath, unsigned int glewActiveTexture);

    unsigned int getTextureID();
};
```

Collision.h

```
#pragma once
#include<glm/glm.hpp>
#include<vector>


struct point
{
        int x;
        int z;
};

//get integer location of given position
point get_location(glm::vec3* cameraPos);

//get nearest obstacles from given position
std::vector<point> get_nearest_obstacles(glm::vec3* cameraPos);
```

```
std::vector<point> end_point_obs(glm::vec3* cameraPos);

bool iscolliding(glm::vec3* cameraPos, point* location);

std::vector<point> findObstacleTowardsPlayer(glm::vec3* enemyPos);

//
==========================================================================
=================
//get_nearest_obstacles() gets array of positions with obstacles around the camearaPos
//go through each element coordinate and determine those that are actually touching
//consider a bounding cube of 0.55f on each side of obstacle coordinates
//find out which direction(x or z)is that obstacle and restrict that coordinate
//if the height is below top of the obstacle
//void collisionTest(glm::vec3* camPos);

void collisionTest(glm::vec3& oldCameraPos, glm::vec3* cameraPos);

//
==========================================================================
=========
// this method is just like obstacle detection method
//after every keyboard movement, a player and an enemy interchange their roles momentarily
//enemy tries to move towards the direction of a player taking small steps
//during the process if it doesn't detect any obstacle along the path
//we conclude that the enemy can see a player
//the process is terminated when the step is so large that it excess or reaches to the boundary of
the maze
//drawback of this approach is that the height of the obstacle doesn't make a difference

bool enemyLineOfSight(glm::vec3 enemyOrigin, glm::vec3 playerPos);
```

Camera.fs

```
#version 330 core
out vec4 FragColor;

struct Light {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoords;

uniform vec3 viewPos;
uniform Light light;

// texture samplers
uniform sampler2D texture1;
uniform sampler2D texture2;
uniform sampler2D texture3;
uniform sampler2D texture4;
uniform sampler2D texture5;

uniform int floor;
uniform int wall;


void main()
{
        vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(light.position - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
        vec3 diffuse = light.diffuse * diff;

        float ambientStrength = 1.0f;
        vec3 ambient = light.ambient * ambientStrength;

        float specularStrength = 0.5;
        vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
        float spec = pow(max(dot(viewDir, reflectDir), 0.0), 64);
    vec3 specular = light.specular * spec * specularStrength;

        if(floor == 0 && wall == 0) FragColor = vec4((ambient + diffuse +
specular)*texture(texture1, TexCoords).rgb, 1.0);
        if(floor == 1 && wall == 0) FragColor = vec4((ambient + diffuse +
specular)*texture(texture2, TexCoords).rgb, 1.0);
        if(floor == 0 && wall == 1) FragColor = vec4((ambient + diffuse +
specular)*texture(texture3, TexCoords).rgb, 1.0);
        if(floor == 1 && wall == 1) FragColor = vec4((ambient + diffuse +
specular)*texture(texture4, TexCoords).rgb, 1.0);
        if(floor == -1 && wall == -1) FragColor = vec4((ambient + diffuse +
specular)*texture(texture5, TexCoords).rgb, 1.0);

}
```

Camera.vs

```
#version 330 core
```

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
        FragPos = vec3(model * vec4(aPos, 1.0));
        Normal = mat3(transpose(inverse(model))) * aNormal;
        TexCoords = aTexCoords;
        gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

Light.fs

```
#version 330 core
out vec4 FragColor;

void main()
{
   FragColor = vec4(0.8f, 1.0f, 0.8f, 1.0f);
}
```

Light.vs

```
#version 330 core
out vec4 FragColor;

void main()
{
   FragColor = vec4(0.8f, 1.0f, 0.8f, 1.0f);
}
```

Bibliography

(n.d.). Retrieved from docs.gl

community. (n.d.). *graphics book*. Retrieved from http://math.hws.edu/graphicsbook

Computer Graphics Learning Materials by Raimond Tunnel, J. J. (n.d.). Retrieved from https://cglearn.codelight.eu/pub/computer-graphics

Hearn, B. &. (2013). Computer Graphics with OpenGL (4th ed.) . In B. &. Hearn, *Computer Graphics with OpenGL (4th ed.)* .

scratchpixel. (n.d.). Retrieved from scratchpixel.com

Vries, J. D. (n.d.). *learnopengl*. Retrieved from learnopengl.com