TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS
DEPARTMENT OF
ELECTRONICS AND COMPUTER ENGINEERING

PROJECT REPORT ON 3D MODELING AND RENDERING OF STADIUM (COMPUTER GRAPHICS)

A COURSE PROJECT PROPOSAL SUBMITTED TO THE DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE PRACTICAL COURSE ON COMPUTER GRAPHICS [EX 554]

Submitted by:

Sandesh Thapa (PUL076BEI036)
Sanim Kumar Khatri (PUL076BEI037)
Suvash Joshi (PUL076BEI045)

Submitted to:

Department of Electronics and Computer Engineering,

Pulchowk Campus Institute of Engineering, Tribhuvan University

Lalitpur, Nepal

2ndyear/II part

2022A.D.

ACKNOWLEDGEMENT

We would like to express our sincere thanks and gratitude to our computer graphics lecturer LNR sir and our lab instructor. Without their constant efforts in guiding and teaching us we would not be able to amass enough knowledge to progress in our project. The materials they provided us really enabled us to level up our level of understanding of computer graphics. This project was a great opportunity to exercise the theoretical teachings of our lecturer.

Also we would like to thank the department of electronics and computer engineering for including these kinds of hands on project so that we can implement what we have learned.

Finally we would like to thank everyone who were involved with this project indirectly and all the learning platforms like YouTube and Google.

ABSTRACT

Computer graphics has become a crucial part of modern day world of information technology. The world is rapidly expanding in terms of better communication and with that the growth in the field of computer graphics has been synonymous. Today, computer graphics is a core technology in digital photography, film, video games, cell phone and computer displays, and many specialized applications.

This project depicts the use of computer graphics and different algorithms used in computer graphics to render a 3D model of a football stadium using OpenGL. The 3D model consists of a stadium on an island surrounded by water from all sides and a mesmerizing view of the mountains in the background.

Additionally the report explores the implementation of these algorithms using C++, OpenGL and GLSL. It is hoped that this report will help all the individuals interested in computer graphics to know more about the use of OpenGL to create and render 3D models using efficient graphics algorithms.

Table of Contents

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATION

2D:	Two Dimensions
3D:	Three Dimensions
OpenGL:	Open Graphics Library
GLSL:	OpenGL Shading Language
GLFW:	Graphics Library Framework
ARB:	Architecture Review Board
CPU:	Central Processing Unit
GPU:	Graphical Processing Unit
COP:	Center of Projection
API:	Application Programming Interface

1. INDRODUCTION

1.1 Background

Our project entitled "3D modelling and rendering of Stadium" is the simulation of football stadium model with the near surroundings in order to complete the objective of this project. Two different views are included: Day view and Night view. Users have the ability to switch between the views by using a keyboard, D for day view and N for night view. Camera is incorporated to navigate the 3D model by using the keys: UP, DOWN, LEFT, RIGHT and the mouse.

1.2 Objectives

The major objectives of this project which is required for fulfilment of practical course on Computer Graphics are listed below:

- To design and render a football stadium.
- To learn about OpenGL for rendering 3d objects.
- TO implement graphics algorithms to create and render real life objects.
- To understand concepts of shaders.
- To implement various lighting models.
- To build teamwork.
- To understand about texture and texture loading in graphics project.

1.3 Scope

This project uses stadium model along with camera component. This designs and implementations are quite handy in game development and animation in the long run Computer graphics is a field of computer science that studies methods for digitally synthesizing and manipulating visual content. Computer graphics deals with generating images with the aid of computers. Today, computer graphics is a core technology in digital photography, film, video games, cell phone and computer displays, and many specialized applications. We have detailed the techniques used in computer graphics to make the objects look more realistic. The theoretical aspects of these algorithms and techniques that are needed to understand the project are well covered in the report.

This project exhibits extensive use of the computer graphics algorithms using OpenGL. We hope that this report will help all the people who read it to understand and generalize the use of computer graphics

2. LITERATURE REVIEW

1. OpenGL

1 OpenGL logo

OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.

2. GLFW

2 GLFW logo

GLFW is a lightweight utility library for use with OpenGL. GLFW stands for Graphics Library Framework. It provides programmers with the ability to create and manage windows and OpenGL contexts, as well as handle joystick, keyboard and mouse input.

3. GLEW

The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

4. GLSL

GLSL is a high level shading language with a syntax based on the C programming language. It was created by OpenGL ARB to give developers more direct control of the graphics pipeline without having to use ARB assembly language or hardware- specific languages.

5. Shaders

Shaders are little programs that rest on the GPU. These programs are run for each specific section of the graphics pipeline. In a basic sense, shaders are nothing more than programs transforming inputs to outputs.
There are mainly two types of shaders and they are:
I. Vertex Shader
II. Fragment Shader

5.1 Vertex Shader

Vertex shaders manipulate coordinates in a 3D space and are called once per vertex. The purpose of the vertex shader is to set up the gl_Position variable — this is a special, global, and built-in GLSL variable. gl_Position is used to store the position of the current vertex.
The void main() function is a standard way of defining the gl_Position variable. Everything inside void main() will be executed by the vertex shader. A vertex shader yields a variable containing how to project a vertex's position in 3D space onto a 2D screen.

5.2 Fragment Shader

Fragment shaders define RGBA (red, green, blue, alpha) colors for each pixel being processed — a single fragment shader is called once per pixel. The purpose of the fragment shader is to set up the gl_FragColor variable. gl_FragColor is a built-in GLSL variable like gl_Position.
The calculations result in a variable containing the information about the RGBA color.

6. Textures

A texture is a 2D image used to add detail to an object. Textures are basically used as images to decorate 3D models and can be used to store many different kinds of data. It is used to add detail to an object. Aside from images, textures can also be used to store a large amount of data to send to the shaders.

7. Lighting

Lighting deals with assigning the color to the surface points of objects. The scene is illuminated with light based upon a lightning model. Lighting model computes the color in terms of intensity values. The luminous intensity or color of a point depends on the properties of the light source, the properties of the surface on which the point lies such as its reflectance, refraction, and the position, orientation of the surface with respect to the light source.

Ambient Light

Ambient light is a simple way to model the combination of the light reflections from various surfaces to produce a uniform illumination. Ambient light has no spatial or directional characteristics. The amount of the ambient light incident is constant for all the surfaces and in all directions. The position of the viewer is not important for modeling the ambient light.
I = IaKa
I: intensity
Ia: intensity of Ambient light
Ka: object's ambient reflection coefficient, 0.0 - 1.0 for each of R, G, and B

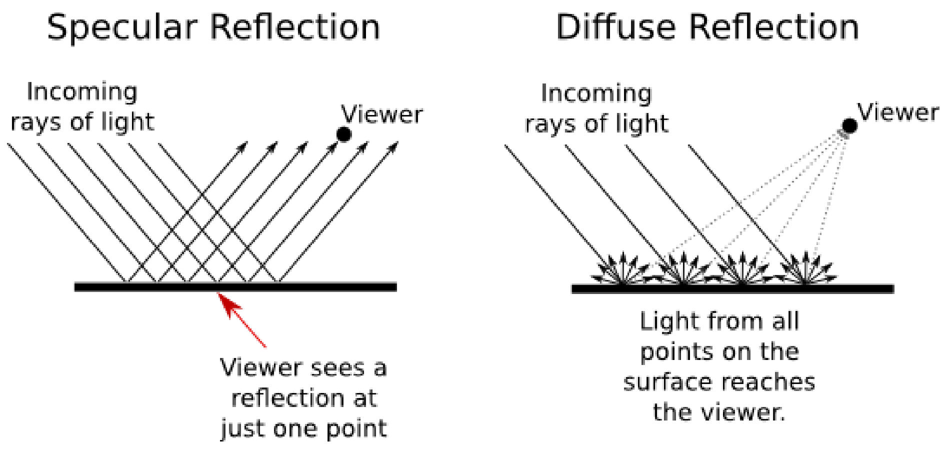
Diffuse Light

Diffuse light is modelled using a point light source. The light comes from a specific direction. This light reflects off the dull surface also. It is reflected with equal intensity in all the directions. The brightness depends upon the angle theta which is the angle between the surface normal and the direction to the light source. The position of the viewer is not considered for diffuse light.

I = Ip Kd cos(theta) or I = Ip Kd(N' * L')
I: intensity
Ip: intensity of point light
Kd: object's diffuse reflection reflection coefficient, 0.0 - 1.0 for each of R, G, and B
N': normalized surface normal
L': normalized direction to light source
*: represents the dot product of the two vectors
It is rare that we have an object in the real world illuminated only by a single light. Even on a dark night there is some ambient light. To make sure all sides of an object get at least a little light we add some ambient light to the diffuse light. The combined equation for the intensity of light considering both the ambient light and diffuse light becomes
I = Ia Ka + Ip Kd(N' * L')
If the light source attenuation factor is also considered .i.e the intensity of the light source varying with the distance then the equation becomes
I = Ia Ka + Fatt Ip Kd(N' * L')

<u>Specular Light</u>
Specular light gives the reflection off the shiny surfaces. The position of the viewer is important in case of the specular reflection. Surfaces with the higher specular component such as metals and plastic give more specular intensity whereas surfaces with lower specular components such as chalk, sand do not reflect much specular light.

I = Ip cos^n(a) W(theta)
I: intensity
Ip: intensity of point light
n: specular-reflection exponent (higher is sharper falloff)
W: gives specular component of non-specular materials
If we put together ambient light, diffuse light and the specular light, the intensity equation becomes
I = Ia Ka + Ip Kd(N' * L') + Ip cos^n(a) W(theta)


Specular Reflection
Incoming rays of light
Viewer
Viewer sees a reflection at just one point
Diffuse Reflection
Incoming rays of light
Viewer
Light from all points on the surface reaches the viewer.

3 Diffuse and Specular reflection

8. Shading Models

The use of the lightning models individually to calculate the intensity of each fragment on the surface of the polygon mesh becomes computationally expensive. For the faster operation, shading models are used which are based upon use of the interpolation for the calculation of the surface intensity.

8.1 Flat Shading

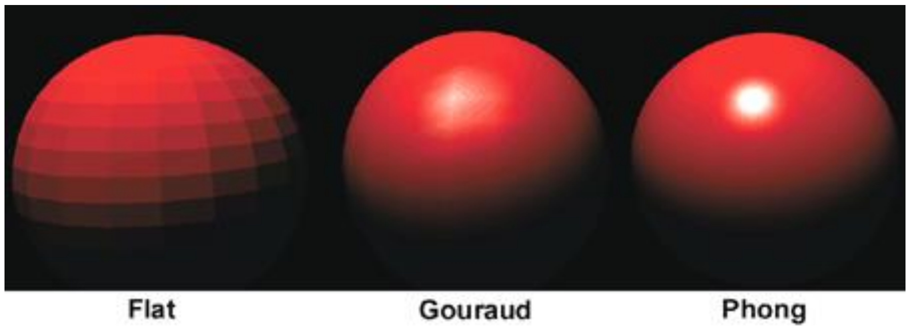
This shading model considers only the ambient light. All the points of the surface have the same intensity when flat shading is applied. It is the fastest shading model. It is applicable only when the object the model is to be used is at a far distance from the light source or the viewer is at far distance.

8.2 Gouraud Shading

In this shading model the average unit normal vector at each vertex of the polygon is calculated. The lightning equation is used at each vertex. It makes use of the linear interpolation to calculate the intensities of the intermediate vertices in the edges. Colors are interpolated across the polygon. The Mach Band effect is caused by the use of Gouraud shading. It can be eliminated by increasing the number of vertices of the polygon. It is slower than the flat shading model.

8.3 Phong Shading

The Phong shading model is similar to the Gouraud shading model. Where Gouraud shading uses normal at the vertices and then interpolates the resulting colours across the polygon, Phong shading goes further and interpolates the normals. Linear interpolation of the normal values at each vertex are used to generate normal values for the pixels on the edges. Linear interpolation across each scan line is used to then generate normals at each pixel across the scan line. Whether we are interpolating normals or colours the procedure is the same. It is better at dealing with the highlights than the Gouraud shading model. It is slower than the Gouraud shading model.



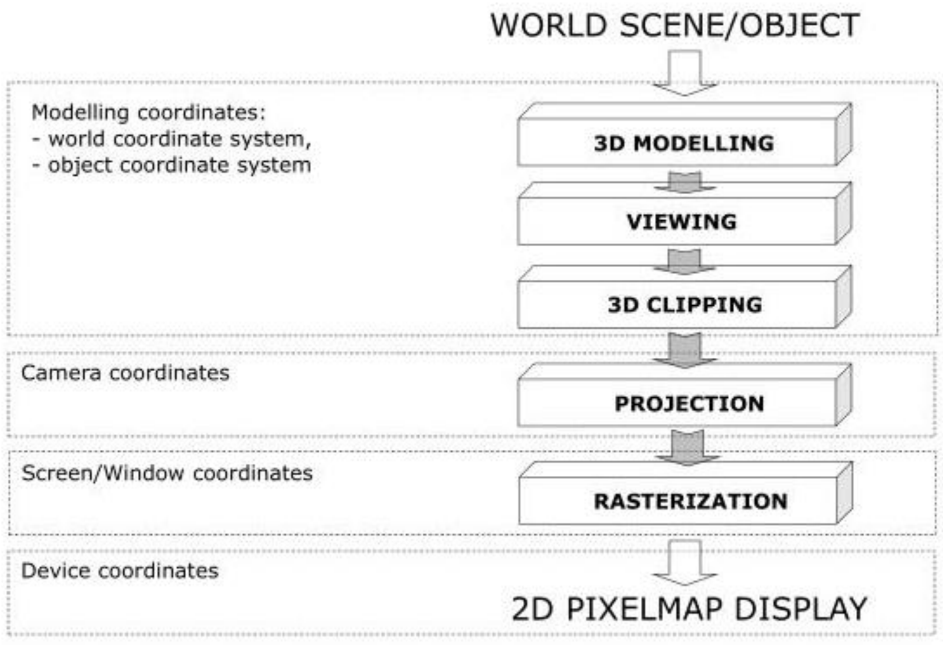
4 Visualization of shading models

9. Coordinate Systems in Computer Graphics

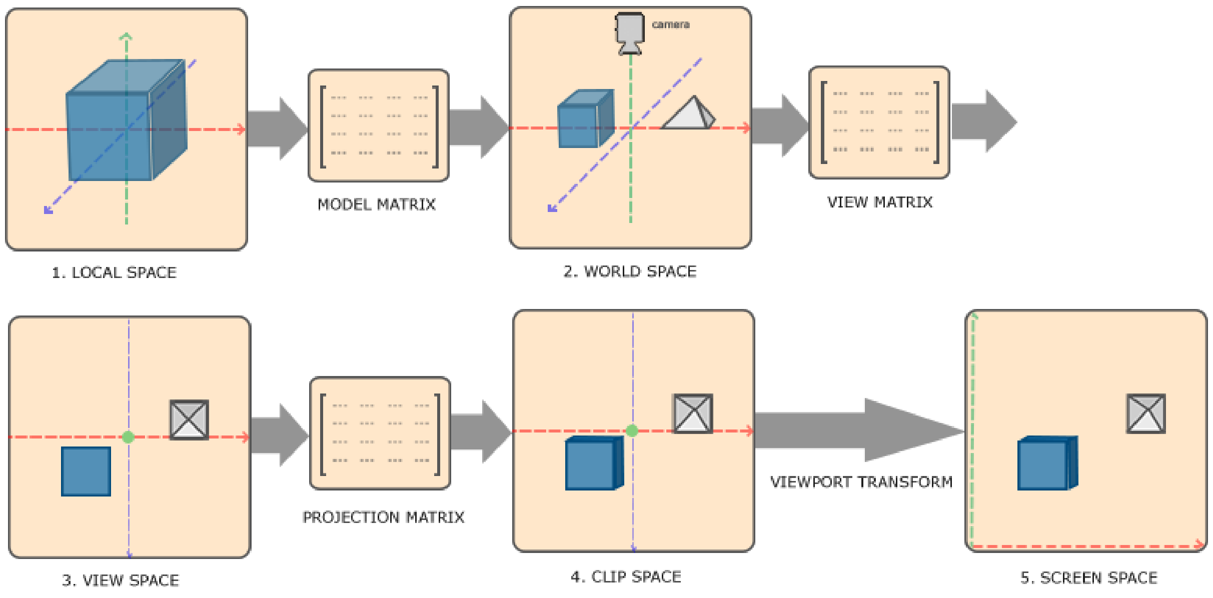
There are 5 coordinate systems that are important in Computer Graphics:

- Local space - original coordinates of the object, relative to object’s origin
- World space - all coordinates relative to a global origin.
- View space (or Eye space)-all coordinates as viewed from a camera's perspective.
- Clip space- all coordinates as viewed from the camera's perspective but with projection applied
- Screen space- all coordinates as viewed from the screen. Coordinates range from 0 to screen width/height

3D Graphics Pipeline



5 *3d Graphics pipelining*



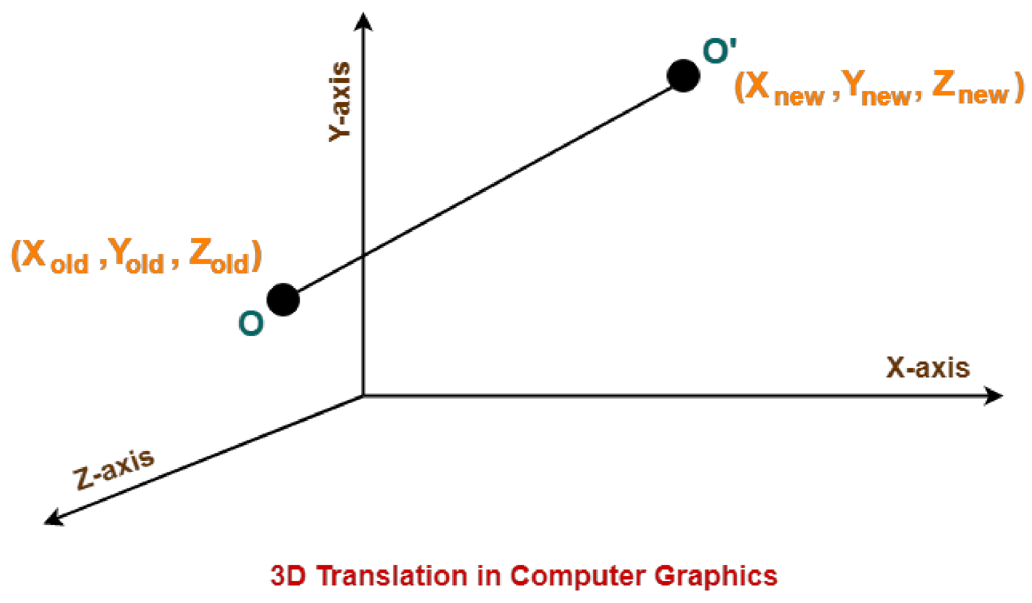
6 *3d Graphics pipelining with matrix*

10. Transformation

All the operation done on 3D objects are transformation, there are two kinds of transformation:
a. Basic transformation (Translation, Scaling, Rotation)
b. Derived transformation (Reflection, Shearing)
The algorithm of the basic transformation are given as follows:

10.1 Translation

In Computer graphics, 3D Translation is a process of moving an object from one position to another in a three dimensional plane.
Consider a point object O that has to be moved from one position to another in a 3D plane.
Let,
Initial coordinates of the object O = (Xold, Yold, Zold)
New coordinates of the object O after translation = (Xnew, Ynew, Zold)
Translation vector or Shift vector = (Tx, Ty, Tz)
Given a Translation vector (Tx, Ty, Tz)-
Tx defines the distance the X-old coordinate has to be moved.
Ty defines the distance the Y-old coordinate has to be moved.
Tz defines the distance the Z-old coordinate has to be moved.


Y-axis
O'
$(X_{new}, Y_{new}, Z_{new})$
$(X_{old}, Y_{old}, Z_{old})$
O
X-axis
Z-axis
3D Translation in Computer Graphics

7 3D Translation

This translation is achieved by adding the translation coordinates to the old coordinates of the object as-

Xnew = Xold + Tx (This denotes translation towards X axis)
Ynew = Yold + Ty (This denotes translation towards Y axis)
Znew = Zold + Tz (This denotes translation towards Z axis)

In Matrix form, the above translation equations may be represented as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

3D Translation Matrix

8 Matric form of 3D translation

10.2 Rotation

In Computer graphics, 3D Rotation is a process of rotating an object with respect to an angle in a three dimensional plane.
Consider a point object O that has to be rotated from one angle to another in a 3D plane.
Let
Initial coordinates of the object O = (Xold, Yold, Zold)
Initial angle of the object O with respect to origin = Φ
Rotation angle = θ

New coordinates of the object O after rotation = (Xnew, Ynew, Znew)

In 3 dimensions, there are 3 possible types of rotation-
X-axis Rotation
Y-axis Rotation
Z-axis Rotation

For X-Axis Rotation

This rotation is achieved by using the following rotation equations Xnew = Xold
Ynew = Yold x cosθ – Zold x sinθ
Znew = Yold x sinθ + Zold x cosθ
In Matrix form, the above rotation equations may be represented as

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

3D Rotation Matrix
(For X-Axis Rotation)

9 Matrix form of 3D X axis rotation

For Y-Axis Rotation

This rotation is achieved by using the following rotation equations-
Xnew = Zold x sinθ + Xold x cosθ
Ynew = Yold
Znew = Yold x cosθ – Xold x sinθ

In Matrix form, the above rotation equations may be represented as

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

3D Rotation Matrix

(For Y-Axis Rotation)

10 Matrix form of 3D Y axis rotation

For Z-Axis Rotation

This rotation is achieved by using the following rotation equations-
Xnew = Xold x cosθ – Yold x sinθ
Ynew = Xold x sinθ + Yold x cosθ
Znew = Zold

In Matrix form, the above rotation equations may be represented as

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

3D Rotation Matrix

(For Z-Axis Rotation)

11 Matrix form of 3D Z axis rotation

10.3 Scaling

In computer graphics, scaling is a process of modifying or altering the size of objects. Scaling may be used to increase or reduce the size of objects. Scaling subjects the coordinate points of the original object to change. Scaling factor determines whether the object size is to be increased or reduced.
If scaling factor > 1, then the object size is increased.
If scaling factor < 1, then the object size is reduced.
Consider a point object O that has to be scaled in a 3D plane.

Let
Initial coordinates of the object O = (Xold, Yold,Zold)
Scaling factor for X-axis = Sx
Scaling factor for Y-axis = Sy
Scaling factor for Z-axis = Sz
New coordinates of the object O after scaling = (Xnew, Ynew, Znew)
This scaling is achieved by using the following scaling equations
Xnew = Xold x Sx
Ynew = Yold x Sy
Znew = Zold x Sz

In Matrix form, the above scaling equations may be represented as

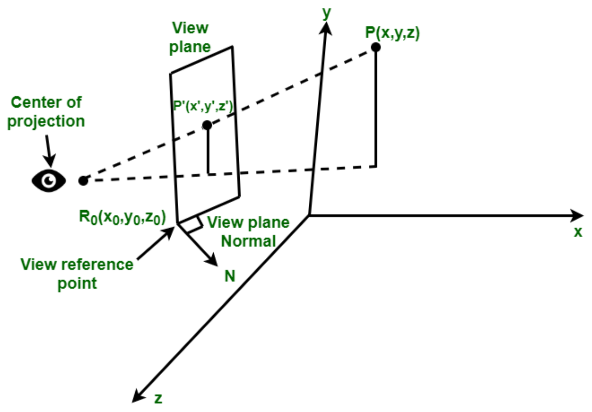
$$\begin{bmatrix} X_{new} \\ Y_{new} \\ Z_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ Z_{old} \\ 1 \end{bmatrix}$$

3D Scaling Matrix

12 Matrix form of scaling

11. Perspective projection

Perspective view gives the realistic representation of the appearance of the 3D object. Object positions are transformed to the view plane along the line that converges to a point called COP. In this projection farther the object from the viewer, small it appears. Two main characteristics of perspective are vanishing points and perspective foreshortening. On the basis of vanishing point, perspective projection is classified as:



13 *Perspective projection*

The perspective projection can be defined in matrix form as well. The matrix form for perspective projection is:

$$\begin{bmatrix} \frac{1}{aspect * \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2 * far * near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

14 Matrix form for perspective projection

12. 3D viewing transformation

The viewing transformation is done to transform the world coordinates to the view-space coordinates in such a way that each object is as seen from the viewer's point of view. It is the first step of the 3D viewing pipeline. The viewing transformation is performed in two steps. First, translate the viewing-coordinate origin at world position P0 = (x0, y0, z0) to the origin of the world coordinate system. The matrix for translating the viewing origin to the world origin is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

15 Translation matrix for viewing transformation

Now, the rotation is performed to align viewing-space coordinate axes with the world coordinate axes. The composite matrix for the rotation is

$$\mathbf{R} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

16 Rotation matrix for viewing transformation

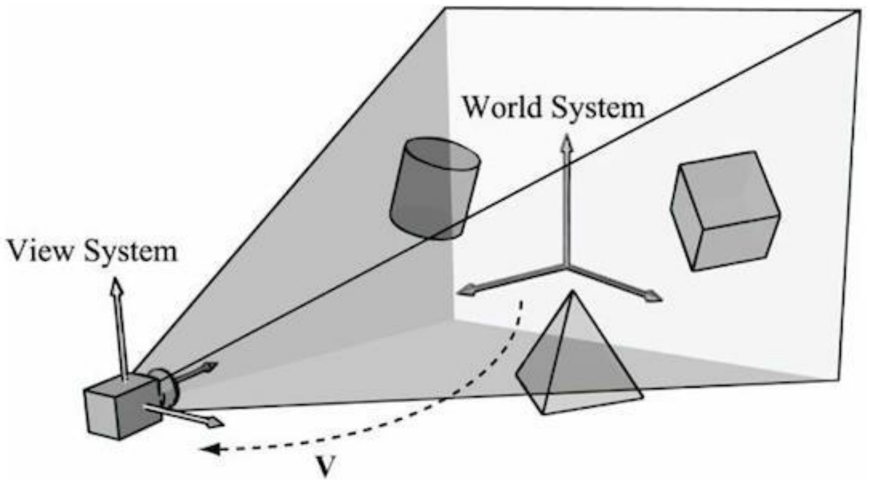
The view-plane normal N defines the direction for the viewing-space z axis and the view-up vector V is used to obtain the direction for its y axis. Using the input values for N and V, we can compute a third vector, U, that is perpendicular to both N and V with a vector cross product. Vector U then defines the direction for the positive x axis of the viewing-space. The unit vectors along the directions U, V and N i.e. u, v and n represent the viewing-space coordinate axes.

$$\mathbf{n} = \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x, n_y, n_z)$$
$$\mathbf{u} = \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V} \times \mathbf{n}|} = (u_x, u_y, u_z)$$
$$\mathbf{v} = \mathbf{n} \times \mathbf{u} = (v_x, v_y, v_z)$$

The final transformation matrix for the transformation from world coordinates to viewing coordinates is obtained as the product of this rotation matrix and translation matrix.

$$\mathbf{M}_{WC,\,VC} = \mathbf{R} \cdot \mathbf{T}$$

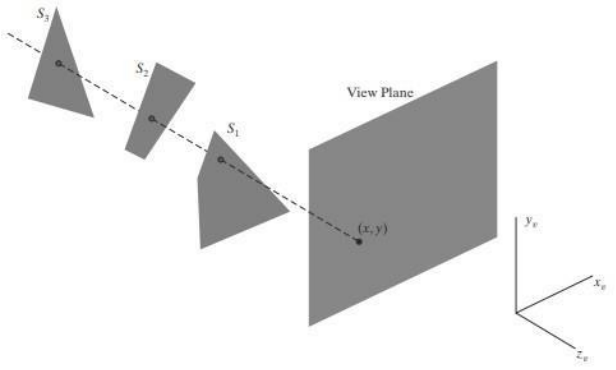
The viewing-space coordinates obtained from the viewing transformation are later transformed to the 2D view plane using projection transformation.



17 Viewing Transformation

13. Depth Test

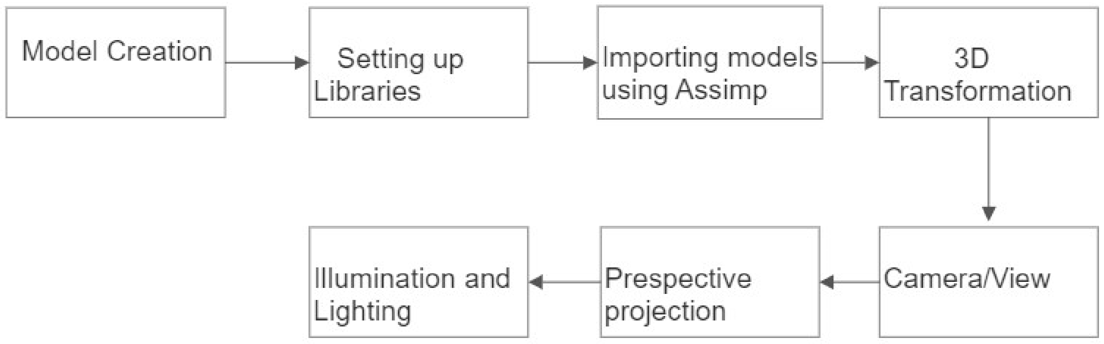
The depth testing is performed using the z-buffer method. It is done in order to detect the visible surfaces so that they are rendered on the screen. In this method, the depth value of each pixel in the projection plane is compared for the detection of the visible surface. The Depth is the distance of the surfaces of polygons in the scene along the z-axis from the view plane.



18 *Z-buffer test*

3. METHODOLOGIES

1. Project Block Diagram



19 Project Block Diagram

1.1 Model Creation

The models used in this project are object files and files with .mtl extension. These were exported from blender files. We tried to create our model from scratch in blender but couldn't due to time complexity, so instead we edited an existing blender file to get our stadium model. Different textures and colors were applied to the model and finally exported to the obj file. This generated obj file contained all the vertices, texture coordinates, normals and faces of the model in triangulated mesh form. Material file (mtl) was also generated from blender which linked various properties of materials.

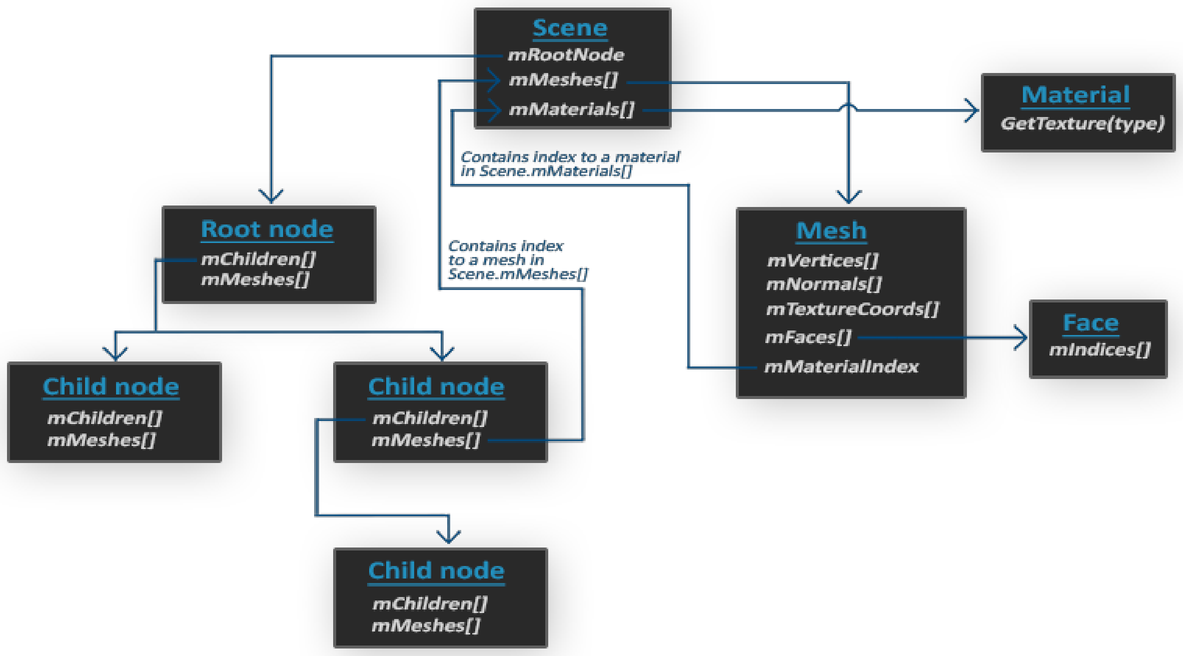
1.2 Setting up Libraries

There are many libraries used in our project for accomplishing specific tasks. The creation and management of windows along with processing user inputs is handled by GLFW. GLEW is used in order to query and load OpenGL extensions. Also, stb_image is used in order to load different textures. All the required libraries required in our project were initialized. A CMake file was created to load these libraries from source code instead of setting binary source code libraries manually for the specific machine. This use of CMake file makes it easy to work in different machines without checking for compatibility and manual setting changes for each machine. The libraries used in our project are listed below:

- GLFW
- GLEW
- GLM
- Assimp
- stb_image

1.3 Importing models using Assimp

The exported model from Blender in obj file format was imported using Assimp which stands for Open Asset Import Library. When importing a model via Assimp it loads the entire model into the Assimp data structure which consists of Scene objects. Assimp then has a collection of nodes where each node contains indices to data stored in the scene object where each node can have any number of children. A model of Assimp's structure is shown below:



20 Assmip Component tree

1.4 Transformation and Projection

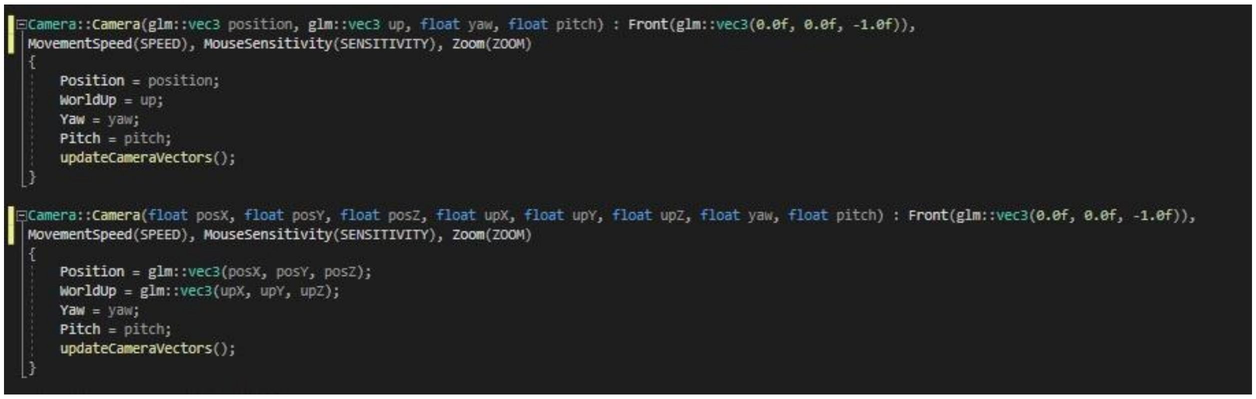
After importing the model, various transformations like translation, scaling, rotation etc. were performed in order to position the model in the required place in 3D space by creating a model matrix. Similarly, the view matrix was created using an instance of Camera class. Finally, the model was projected to the screen using perspective projection in order to obtain a real world viewing experience by defining a projection matrix. All these matrices were combined and multiplied with the position of the object in the vertex shader to obtain the required realistic view.

1.5 Camera

In order to obtain a realistic view of the object, camera position can be changed by pressing keys or moving the mouse position. These positions are sensed and camera position is changed accordingly for changing the view of an object.

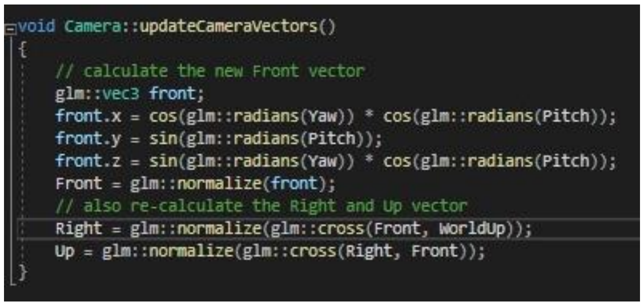
1.6 Lighting

Many different types of point lights are positioned in different positions by defining coordinates of the light source. For object color to produce ambient lighting, object color, light color and ambient strength was multiplied. For diffuse lighting, the lighting changes with angle. The dot product of normal and light coordinate vectors was multiplied to the object color. For specular lighting, object color changes with view position and the lighting intensity is obtained by multiplying color with dot product of reflection and the view direction. There are three illumination model we can use: Ambient lighting, Diffuse reflection and Specular reflection. In this lighting model we used basic illumination model i.e. Ambient+Diffuse+Specular model for better result.



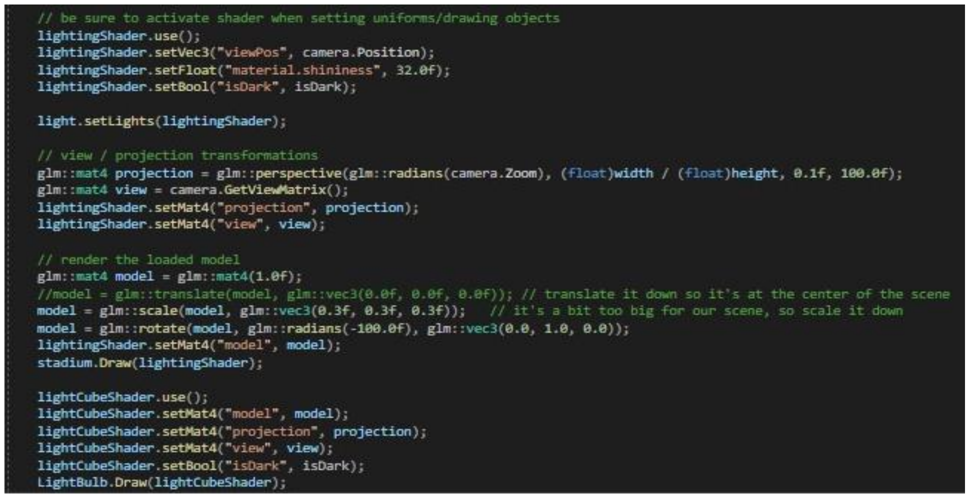
```cpp
Camera::Camera(glm::vec3 position, glm::vec3 up, float yaw, float pitch) : Front(glm::vec3(0.0f, 0.0f, -1.0f)),
MovementSpeed(SPEED), MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
{
    Position = position;
    WorldUp = up;
    Yaw = yaw;
    Pitch = pitch;
    updateCameraVectors();
}

Camera::Camera(float posX, float posY, float posZ, float upX, float upY, float upZ, float yaw, float pitch) : Front(glm::vec3(0.0f, 0.0f, -1.0f)),
MovementSpeed(SPEED), MouseSensitivity(SENSITIVITY), Zoom(ZOOM)
{
    Position = glm::vec3(posX, posY, posZ);
    WorldUp = glm::vec3(upX, upY, upZ);
    Yaw = yaw;
    Pitch = pitch;
    updateCameraVectors();
}
```

21 Code snippet to initialize camera component



```cpp
void Camera::updateCameraVectors()
{
    // calculate the new Front vector
    glm::vec3 front;
    front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
    front.y = sin(glm::radians(Pitch));
    front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
    Front = glm::normalize(front);
    // also re-calculate the Right and Up vector
    Right = glm::normalize(glm::cross(Front, WorldUp));
    Up = glm::normalize(glm::cross(Right, Front));
}
```

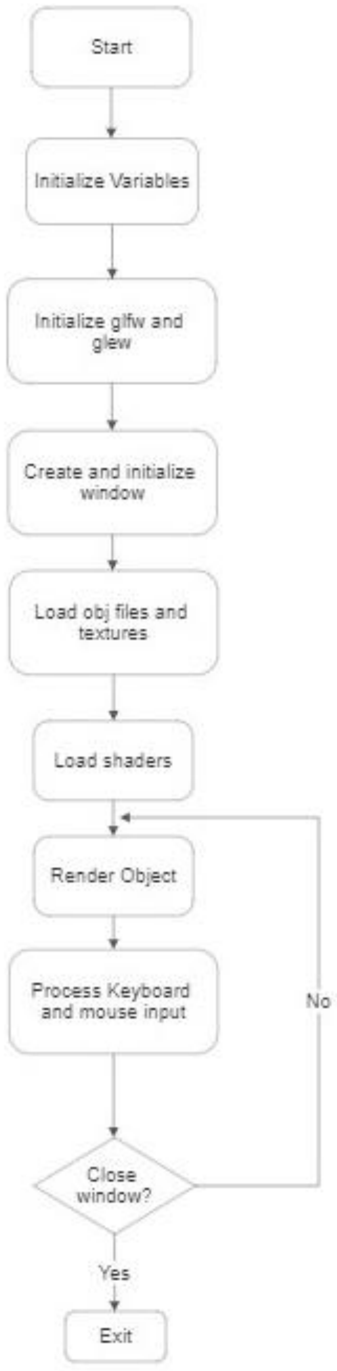
22 Code snippet to update camera vectors



```cpp
    // be sure to activate shader when setting uniforms/drawing objects
    lightingShader.use();
    lightingShader.setVec3("viewPos", camera.Position);
    lightingShader.setFloat("material.shininess", 32.0f);
    lightingShader.setBool("isDark", isDark);

    light.setLights(lightingShader);

    // view / projection transformations
    glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)width / (float)height, 0.1f, 100.0f);
    glm::mat4 view = camera.GetViewMatrix();
    lightingShader.setMat4("projection", projection);
    lightingShader.setMat4("view", view);

    // render the loaded model
    glm::mat4 model = glm::mat4(1.0f);
    //model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.0f)); // translate it down so it's at the center of the scene
    model = glm::scale(model, glm::vec3(0.3f, 0.3f, 0.3f));   // it's a bit too big for our scene, so scale it down
    model = glm::rotate(model, glm::radians(-100.0f), glm::vec3(0.0, 1.0, 0.0));
    lightingShader.setMat4("model", model);
    stadium.Draw(lightingShader);

    lightCubeShader.use();
    lightCubeShader.setMat4("model", model);
    lightCubeShader.setMat4("projection", projection);
    lightCubeShader.setMat4("view", view);
    lightCubeShader.setBool("isDark", isDark);
    LightBulb.Draw(lightCubeShader);
```

23 Code snippet to update matrices and load models

2. Flow Chart



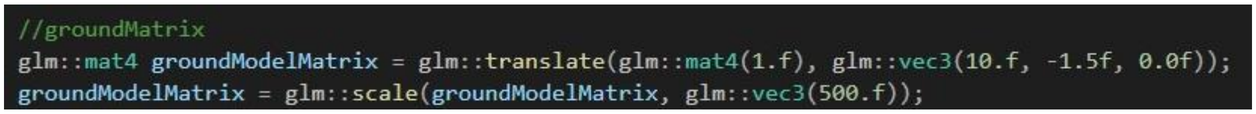
24 Flow chart

3. Algorithm and Implementation

3.1 Transformation

All the operation done on 3D objects are transformation, there are two kinds of transformation:
a. Basic transformation (Translation, Scaling, Rotation)
b. Derived transformation (Reflection, Shearing)
The algorithm for transformation area as mentioned in *Literature review* section *10. Transformation* where all the algorithm and implementation of the translation, scaling and rotation are mentioned along with matrix implementation.

In this project matrices and vectors along with their operations are carried out with the help of GLM library. A snippet for transformation used in the project is given below:

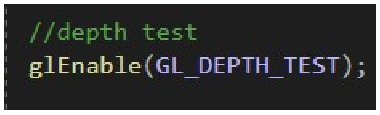


```
//groundMatrix
glm::mat4 groundModelMatrix = glm::translate(glm::mat4(1.f), glm::vec3(10.f, -1.5f, 0.0f));
groundModelMatrix = glm::scale(groundModelMatrix, glm::vec3(500.f));
```

25 Code snippet for transformation

3.2 Depth test

The algorithm is implemented using a Depth buffer, to store the depth value of the fragments and a fragment buffer, to store the color values for storing the color of the fragments. The depth buffer is automatically created by the windowing system to store the depth values and initialized to 1.0 for all the screen fragments. Depth testing is enabled in OpenGL calling the function glEnable with flag GL_DEPTH_TEST. The OpenGL maintains the depth values of the fragments in the depth buffer. During the rendering process, OpenGL compares the z-value of the fragments with the contents of the depth buffer. The fragment is rendered and its depth value is updated in the depth buffer whenever the fragment passes the depth test else the fragment is discarded. The Depth buffer is cleared before rendering each frame.



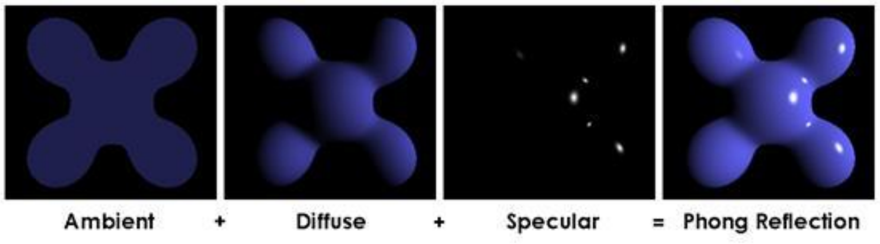
```
//depth test
glEnable(GL_DEPTH_TEST);
```

26 Code snippet for depth test

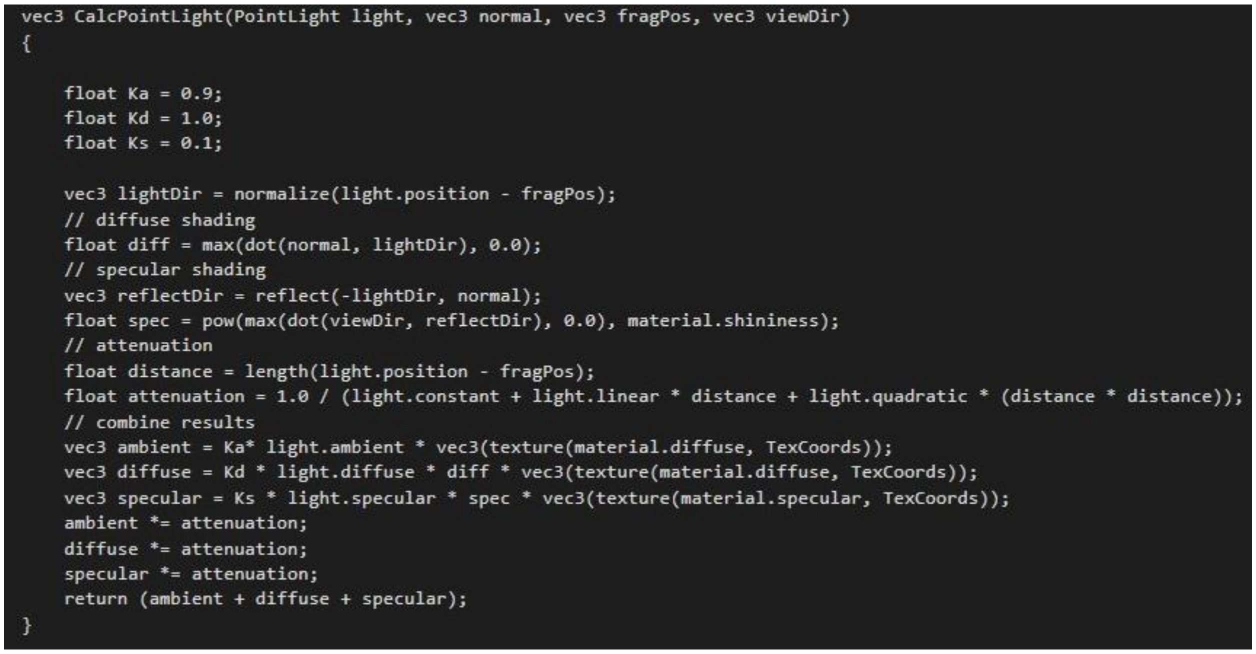
3.3 Illumination model

The illumination model used in this project is **Phong reflection model.** Phong shading may also refer to the specific combination of Phong interpolation and the Phong reflection model, which is an empirical model of local illumination. It describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces. It is based on Bui Tuong Phong's informal observation that shiny surfaces have small intense specular highlights, while dull surfaces have large highlights that fall off more gradually. The reflection model also includes an ambient term to account for the small amount of light that is scattered about the entire scene.

$$I = I_a k_a + f_{att} I_p [k_d (\overline{N}.\overline{L}) + k_s (\overline{V}.\overline{R})^n]$$



27 Phong illumination model



```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
{

    float Ka = 0.9;
    float Kd = 1.0;
    float Ks = 0.1;

    vec3 lightDir = normalize(light.position - fragPos);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // attenuation
    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic * (distance * distance));
    // combine results
    vec3 ambient = Ka* light.ambient * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse = Kd * light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
    vec3 specular = Ks * light.specular * spec * vec3(texture(material.specular, TexCoords));
    ambient *= attenuation;
    diffuse *= attenuation;
    specular *= attenuation;
    return (ambient + diffuse + specular);
}
```

28 Code snippet for Phong reflection model

4. LANGUAGES AND TOOLS

Languages used:

- C++
- GLSL

Libraries used:

- OpenGL
- GLFW
- GLEW
- Assimp
- GLM
- stb_image

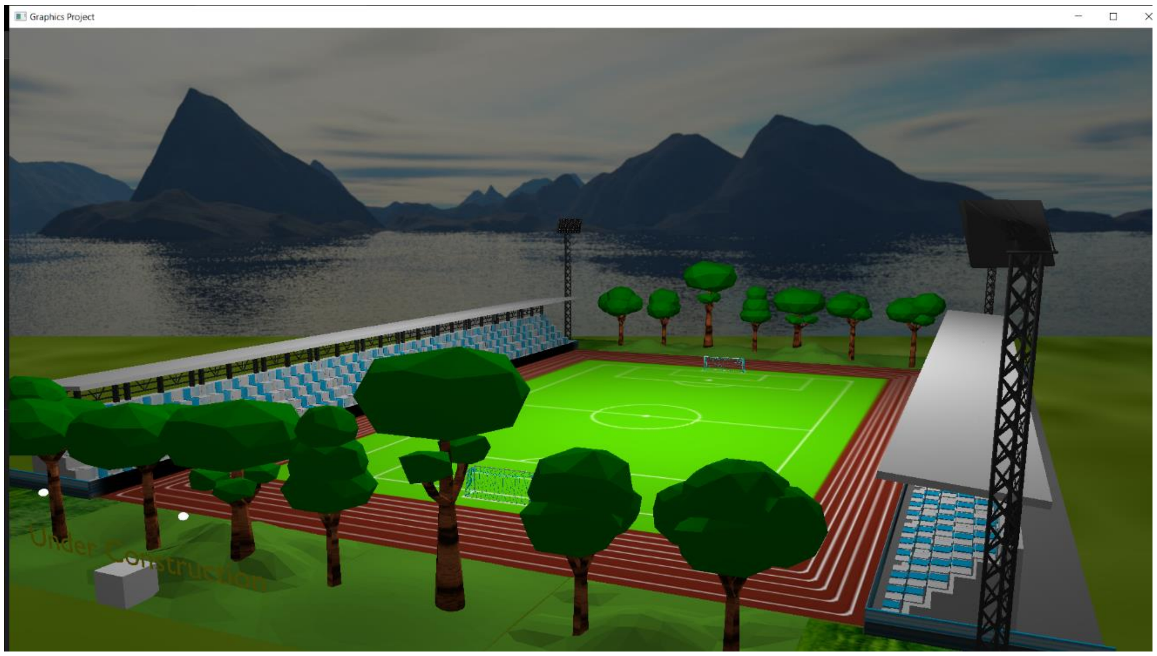
Tools used:

- Visual Studio 2019
- Blender
- Notepad++

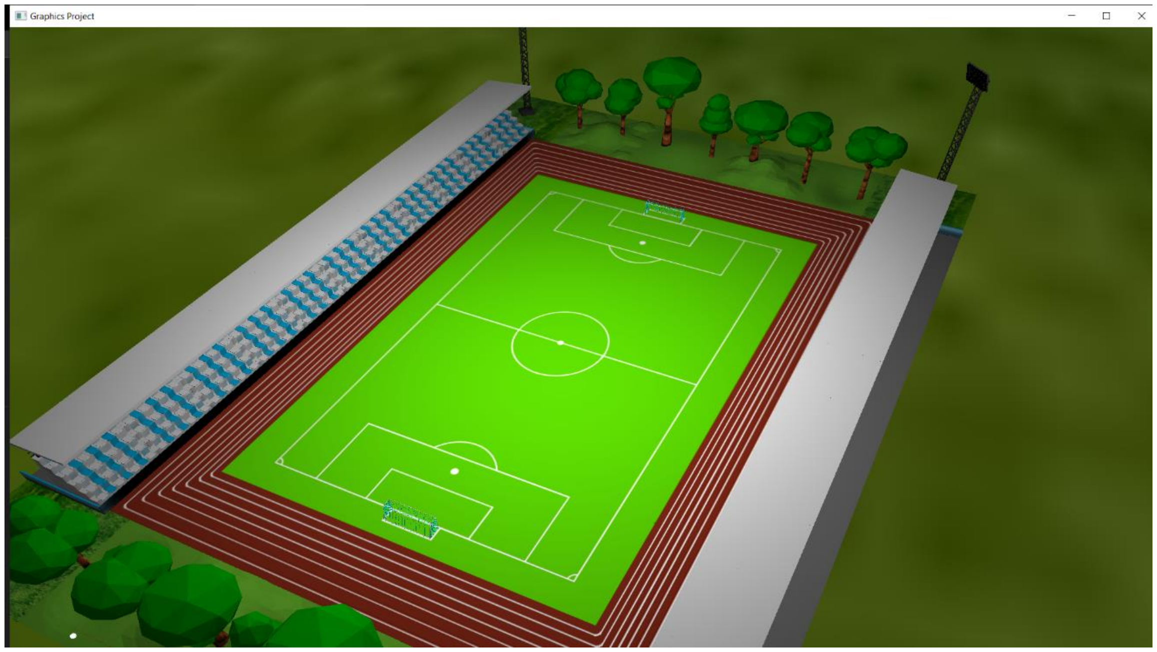
5. RESULT

Graphics Project
Under Construction

29 Night view 1



30 Day view 1



31 Day view 2

Graphics Project
Under Construction

32 Night view 2

Graphics Project

33 Day view 3

Graphics Project

34 Day view 4

Graphics Project

35 Day view 5

6. CODES

The codes of this project has been uploaded to github repository along with all the files used to make this project work. The github repo link for this project is:

https://github.com/SanimKumarKhatri/Computer_Graphics_project.git

7. LIMITATIONS AND FUTURE WORKS

The application currently features a fly style camera for moving freely around the scene. It can move across the objects in the scene and realism is missing. Collision detection algorithms can be used to avoid it and make it more realistic. The meshes for the scenes modeled using blender have a large number of vertices. No checking of the double vertices has been done. Thus, presence of the redundant vertices might have hampered the performance causing lags. We can redesign the same models eliminating the redundant vertices.

The current application is designed with the use of OpenGL API. Better graphics API such as Vulkan can be used for designing the application with faster performance and better image quality at low overhead. Other future enhancements include redesigning the application as a 3D game with the addition of the characters, levels, physics, audio, First person camera, etc. The scene in the application can also be used in other open world games.

8. CONCLUSIONS

In conclusion, we would like to state that this project has helped us a lot in developing a clear sense of understandings of different algorithms used in computer graphics and hence achieved the goal of this project. Through this model rendering program we were able to implement all the different aspects of computer graphics like lighting, transformation etc. these type of models can be useful in rendering different things that cannot be studied by human eye and also visualize them accordingly.

Also, we would like to state that the use of computer graphics is ever growing and it is likely to grow more in the coming days.

9. REFERENCES

1. Donald D. Hearn and M. Pauline Baker, "Computer Graphics C version (2nd edition)"
2. Learn OpenGL, extensive tutorial resource for learning Modern OpenGL https:://learnopengl.com
3. OpenGL YouTube playlist by The Cherno YouTube playlist: https://youtu.be/W3gAzLwfIP0
4. Documentation page for OpenGL docs.gl