TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

A project report on: Computer Graphics

Stadium

Submitted by:

Kisan Bista	074BEX413
Prashant Ghimire	074BEX426
Praveen Gautam	074BEX431
Sushil Raj Regmi	074BEX446

Submitted to:

Dr. Basanta Joshi
Department of Electronics
and
Computer Engineering

ACKNOWLEDGMENT

We would like to express our deepest appreciation to all those who provided us the possibility to complete this project on computer graphics. First of all, we owe our deep gratitude to our respected sir, Dr. Basanta Joshi whose contribution in simulating suggestions and encouragement, helped us to coordinate project writing this report. We are thankful to Mr. Suresh sir, our instructor, who inspires us constantly for making a project. We would also like to thank The Department of Electronics and Computer Engineering, Pulchowk Campus for providing us with this platform to make a project. We students learned a lot about generation of computer graphics And it's current status.
We are thankful to and fortunate enough to get constant encouragement, support and guidance from all of our friends who helped us in successfully completing our Project.

ABSTRACT

Computer Graphics is always being used to model objects and architectures, apply required texture and color to them and observe them simulating the real-world scenario. This has made designers very easy to create their projects and deal with clients. This has also helped clients a lot to ask for their needs before the actual project is done benefiting both designers and clients.

This simulation of real-world scenario of the architectural model is in high demand now a days and our project is just a small imitation to this. We have coded using C++ language and a cross platform library called OpenGL to interact with GPU. The use of OpenGL simplified our project a lot.

The main idea behind our project is to simulate the Wembley football stadium of London and apply lighting and viewing effect to it. The model of the stadium is first created in blender and then the waveform obj object created by it is imported in OpenGL to apply various effects.

Table of Contents

1. INTRODUCTION

1.1. Background

Any image taken by a camera or that displayed in the screen is 2D. To add more realistic features to any image, graphics can be used. For this, the programming should be done in Graphics Processing Unit. Various algorithms and various concepts have been used in this project to create a 3D image.

1.2. Objective

The main objective of this project is to model Wembley Stadium.

1.3. Problem Overview

As the main goal of our project, we are to create a model of a stadium. This has been accomplished by creating a 3d model of the stadium in blender and export waveform obj, material and texture from it and import it in OpenGL and apply transformation, projection, camera movements and lighting using this library.

2. THEORETICAL BACKGROUND

2.1. Blender

Blender is the free and open source 3D creation suite. It supports the entirety of the 3D pipeline—modeling, rigging, animation, simulation, rendering, compositing and motion tracking, video editing and 2D animation pipeline.
It can be used to create complex models and extract the vertices, normal coordinates, texture coordinates and faces in waveform obj type of these models. This data can later be used to obtain the model data in OpenGL.

UV mapping is the 3D modelling process of projecting a 2D image to a 3D model's surface for texture mapping. In Blender, UV mapping is done to extract texture and color from the created model to later use it in OpenGL.

MTL is a data directory which contains examples of MTL files. An MTL file is an auxiliary file containing definitions of materials that may be accessed by an OBJ file. The OBJ file must specify the name of the MTL file by a command. It stores the properties of the materials to be used along with obj file.

2.2. OpenGL

OpenGL is mainly considered an API (an Application Programming Interface) that provides us with a large set of functions that we can use to manipulate graphics and images. However, OpenGL by itself is not an API, but merely a specification, developed and maintained by the Khronos Group.

When developing in immediate mode, in old days, most of the functionality was hidden in the library and developers didn’t have much freedom at how OpenGl does its calculations which was an easy-to-use method for drawing graphics. When developing in core-profile which is a modern practice that is very flexible (provides developers to excess GPU and how programs work) and efficient but a bit difficult to learn than the immediate mode.

2.2.1. Creating a window:

The first thing we need to do before we start creating stunning graphics is to create an OpenGL context and an application window to draw in. However, those operations are specific per operating system and OpenGL purposefully tries to abstract itself from these operations. This means we have to create a window, define a context and handle user input all by ourselves. Here we will be using GLFW libraries that already provide the functionality we seek.

2.2.2. GLFW:

GLFW is a library, written in C, specifically targeted at OpenGL. GLFW gives us the bare necessities required for rendering goodies to the screen. It allows us to create an OpenGL context, define window parameters, and handle user input, which is plenty enough for our purposes.

2.2.3. GLEW:

It stands for OpenGL Extension Wrangler Library. It is a cross-platform C/C++ library that helps in querying and loading OpenGL extensions. It provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

2.2.4. GLSL:

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

2.2.5. GLM:

OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications. GLM provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL so that anyone who knows GLSL, can use GLM as well in C++.

This project isn't limited to GLSL features. An extension system, based on the GLSL extension conventions, provides extended capabilities: matrix transformations, quaternions, data packing, random numbers, noise, etc...

2.2.6. Shader:

A shader is a type of computer program that was originally used for shading. Shaders are also very isolated programs in that they're not allowed to communicate with each other. The only communication they have is via their inputs and outputs. There are two types of shaders used and they are:

1. Fragment shader
2. Vertex shader

2.2.6.1. Fragment shader:

A fragment shader is a piece of code that is executed once, and only once, per fragment which are responsible for painting each primitive's area. The minimum task for this shader is to output an RGBA color.

2.2.6.2 Vertex shader:

Vertex shader provides the vertex positions to clip coordinates, to take us to the next stage. They are oriented to the scene geometry and can-do things like adding cartoony silhouette edges to objects, etc.

2.1.7. Textures:

Textures are typically used for images to decorate 3D models, but in reality, they 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 collection of data to send to the shaders.

2.3. Camera:

When we're talking about camera, we're talking about all the vertex coordinates as seen from the camera's perspective as the origin of the scene: the view matrix transforms all the world coordinates into view coordinates that are relative to the camera's position and direction. To define a camera, we need it position in world space, the direction it's looking at, a vector pointing to the right and a vector pointing upwards from the camera. We're actually going to create a coordinate system with 3 perpendicular unit axes with the camera's position as the origin. The basic things to define the camera are as enlisted below:

2.3.1. Camera Position:

The camera position is basically a vector in world space that points to the camera’s position.

2.3.2. Camera Direction:

The next vector required is the vector containing the information about the direction in which the camera is pointing at. If we define a vector which contains the points where we want to look at, then the difference of the camera position with the look-at vector gives the direction of the camera. Since in the left-handed system the direction in which camera looks is negative direction. So, this vector also defines our view coordinate system z-axis.

2.3.3. Right Axis:

The next vector to define the camera information is the right vector which represents the x-axis of the camera space. To calculate the right axis, we take the help of the up-vector (a vector that points towards up in world space). Then a cross product between the up vector and the direction vector gives the right axis vector.

2.3.4. Up Axis:

Now after getting the x-axis and the z-axis vector the vector pointing in the up direction (Up-Axis) is calculated from the cross product of the two vectors. It defines the y-axis of the camera/view space or our View Coordinate System.

2.3.5. Look-At Matrix:

The look-at matrix creates a view matrix that looks at a given target. The look-at matrix contains three perpendicular axis and the position vector to define the camera/view space. The above look-at matrices contains **R** (which is the right vector),**U** (up-vector),**D** (the direction vector) and **P** (the position vector).So a look-at matrix defines the complete coordinate system for the camera in the Opengl. While using this in OpenGL we only have to specify the a camera position, a target position and a vector that represents the up vector in world space. GLM then creates the Look-At matrix that we can use as our view matrix.

2.4. Lighting:

Lighting in the real world is extremely complicated and depends on way too many factors, something we can't afford to calculate on the limited processing power we have. Lighting in OpenGL is therefore based on approximations of reality using simplified models that are much easier to process and look relatively similar. These lighting models are based on the physics of light as we understand it.

2.4.1. Ambient Lighting:

Light usually does not come from a single light source, but from many light sources scattered all around us, even when they're not immediately visible. One of the properties of light is that it can scatter and bounce in many directions, reaching spots that aren't directly visible; light can thus *reflect* on other surfaces and have an indirect impact on the lighting of an object. It is a simplistic model of global illumination. Adding ambient lighting to the scene is really easy. We take the light's color, multiply it with a small constant ambient factor, multiply this with the object's color, and use that as the fragment's color.

2.4.2. Diffuse Lighting:

Ambient lighting by itself does not produce the most interesting results, but diffuse lighting will start to give a significant visual impact on the object. Diffused light is a soft light with neither the intensity nor the glare of direct light. It is scattered and comes from all directions. Thus, it seems to wrap around objects. It is softer and does not cast harsh shadows. Diffuse lighting gives the object more brightness the closer its fragments are aligned to the light rays from a light source.

The sun is the only source of light present in the simulation. So every illumination is dependent to it. The sun acts as a diffused lighting source. The working of this method is described below in brief.

2.4.3. Specular Lighting:

Similar to diffuse lighting, specular lighting is based on the light's direction vector and the object's normal vectors, but this time it is also based on the view direction e.g. from what direction the player is looking at the fragment. Specular lighting is based on the reflective properties of surfaces. If we think of the object's surface as a mirror, the specular lighting is the strongest wherever we would see the light reflected on the surface.

2.5 Model Loading using ASSIMP

Open Asset Import Library (short name: ASSIMP) is a portable Open Source library to import various well-known 3D model formats in a uniform manner. The most recent version also knows how to export 3d files and is therefore suitable as a general-purpose 3D model converter. See the feature-list. The imported data is provided in a straightforward, hierarchical data structure. Configurable post processing steps (i.e., normal and tangent generation, various optimizations) augment the feature set.

3. METHODOLOGY

3.1. Creating model in Blender:

Firstly, a blender model of stadium was created using the meshes and curves using all the available functions. This model was colored using material coloring and then the model was exported to obj file. This obj file contained vertices, texture coordinates, normal and faces in triangulated mesh form. The material file which linked various properties of materials was also exported form blender and by UV mapping the colors applied in the materials was extracted to a jpg file.

3.2. Initialization of Required Libraries:

The project used GLEW for querying and loading OpengGL extensions, glfw to create and manage windows and OpenGL contexts as well as use input devices and stb_image to load texture images. GLM, a math library was used for mathematical calculations and operations.

3.3. Importing Model:

The vertices, texture coordinates and face coordinates were loaded into ASSIMP database. ASSIMP imports the data from obj file and stores into its database. The vertices and texture coordinates of different meshes of the model will be loaded into the mesh object created by ASSIMP by load model. The coordinated are plotted using the draw function which takes shader as the argument. Also the texture was imported using stb_image library.

3.4 Transformation and Projection:

The model was projected to the screen using perspective projection to obtain the real-world viewing experience by using the glperspective function. And the required view was obtained by multiplying the transformation functions with the position of the plot.

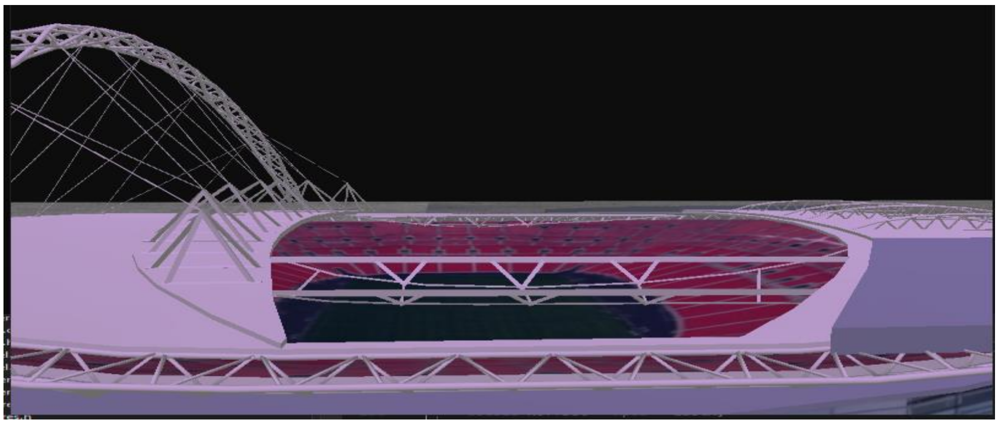
3.5 Camera:

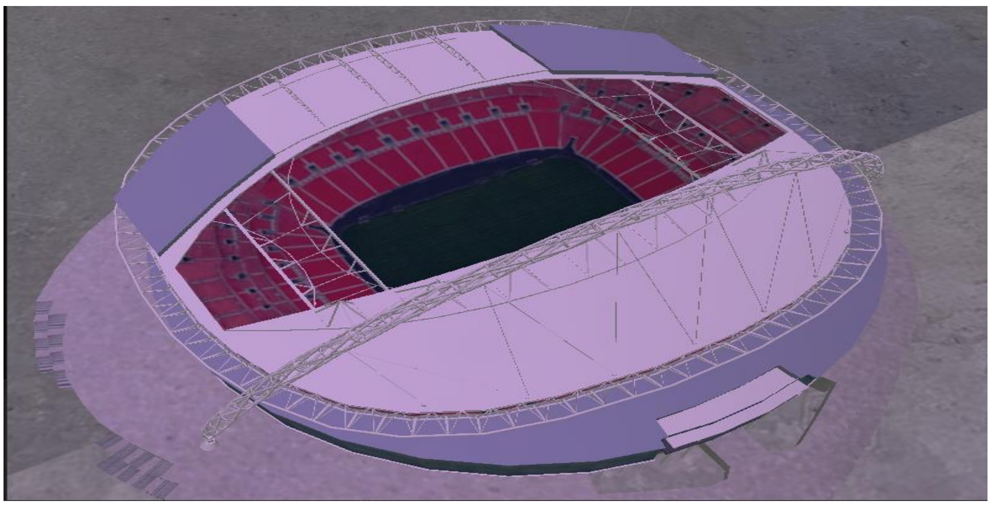
The camera position is used to change the view of the object. The pressing of keys and mouse movement is first sensed and the camera position is changed accordingly. The keyboard presses change camera position up, down and sidewise, while the mouse movements change the yaw and pitch. This pressing is sensed and the sensitivity and velocity is multiplied in the camera position accordingly to obtain the required camera angle.

3.6 Lighting:

The coordinates for light and the ambient strength was defined.
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 vector was multiplying 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.

4. RESULT:





5.

LIMITATIOS:

We faced a lot of obstacles while doing the projects and this led the project to be limited. First, we had planned to create a cricket stadium. We had created a blender model of the cricket stadium and exported to obj file and the file was also loaded. But we couldn't UV map the model to obtain the texture image file which forced us to change the model to a football stadium. All the light types were also not used. Also, the effects applied were limited due to limitation of time.

6. CONCLUSION

Thus, the 3d model was created using Blender, it was exported to obj file and was later imported in the program. Various lights and camera effects were applied in the model to using GLEW, GLM, GLFW libraries.

7. REFERENCES

https://www.khronos.org/opengl/
wiki/ http://www.opengl-
tutorial.org/
https://learnopengl.com/
http://ogldev.atspace.uk/
http://sonarsystems.uk/
https://open.gl/