A

PROJECT REPORT

ON

3D MODELLING OF DINING ROOM

SUBMITTED BY:

AMRIT PUN (PUL074BEX003)
PAWAN POKHREL (PUL074BEX017)
PAWAN SAPKOTA SHARMA (PUL074BEX018)
PRATEEK PUDASAINEE (PUL074BEX027)

SUBMITTED TO:
DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

FALGUN 22, 2076

Acknowledgment

The sensation and final upshot of this project required a lot of guidance and assistance from many people and we are exceptionally honored to have got this all along the completion of our project.

We would like to express our sincere gratitude to our lecturer Dr. Basanta Joshi and lab assistant Mr. Suresh Pokhrel Sir, for providing such a splendid opportunity to work on this project and giving us all support and guidance which made us accomplish the project's goal duly.

Abstract

This project work was given to us as a project for our academic session B.E. (Electronics and Communication) Third Year First Part as prescribed in the syllabus designed by IOE, TU. The main aim of this project was to develop a good understanding of the computer graphics fundamentals, especially 3D rendering. For this purpose, The Dining Room, IOE was modeled and rendered. Its modelling was done in Blender. The Blender model was exported as a *.fbx* file, a binary file containing different information about vertices of the model like position, normals, tangents, bi-tangents, etc. Textures were also exported along with the above binary model file in the form of png or jpg image, which contains information about the lighting maps, a technique of getting the diffusion and specular coefficient for different materials of the model. All the textures and models were loaded in the OpenGL program using third-party libraries Assimp and stbi_image respectively and hence were rendered using suitable lighting and camera transform models. All the basic theories of Computer Graphics about 3D rendering prescribed by the syllabus were applied in the project. There were many challenges which were faced during the modeling, loading and simulating.

Table of Contents

Objectives

The major objectives of the project are listed below:

1. To learn 3D rendering using the various mathematical concepts of Computer Graphics.
2. To learn about Modern OpenGL specification and how they work.
3. To know about different stages of Graphics pipelining and their importance in rendering.
4. To learn what *shaders* are and to write shaders for different stages of pipelining involved.
5. To optimize a program in terms of time and space up to the greatest extent as possible.
6. To make us able to work on major projects in the coming future.
7. To improve the ability to work and cooperate in a team.

Introduction

OpenGL

OpenGL is mainly considered an API (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.

According to *Khronos Group* themselves, “OpenGL® is the most widely adopted 2D and 3D graphics API in the industry, bringing thousands of applications to a wide variety of computer platforms. It is window-system and operating-system independent as well as network-transparent. OpenGL enables developers of software for PC, workstation, and supercomputing hardware to create high-performance, visually compelling graphics software applications, in markets such as CAD, content creation, energy, entertainment, game development, manufacturing, medical, and virtual reality. OpenGL exposes all the features of the latest graphics hardware.”

The OpenGL specification specifies exactly what the result/output of each function should be and how it should perform. It is then up to the developers implementing this specification to come up with a solution of how this function should operate. Since the OpenGL specification does not give us implementation details, the actual developed versions of OpenGL are allowed to have different implementations, as long as their results comply with the specification (and are thus the same to the user).

The people developing the actual OpenGL libraries are usually the graphics card manufacturers. Each graphics card that is bought supports specific versions of OpenGL which are the versions of OpenGL developed specifically for that card (series). When using an Apple system the OpenGL library is maintained by Apple themselves and under Linux there exists a combination of graphic suppliers' versions and hobbyists' adaptations of these libraries. This also means that whenever OpenGL is showing weird behavior that it shouldn't, this is most likely the fault of the graphics card manufacturers or, whoever developed or maintained the library.

OpenGL is by itself a large state machine: a collection of variables that define how OpenGL should currently operate. The state of OpenGL is commonly referred to as the OpenGL context. When using OpenGL, we often change its state by setting some options, manipulating some buffers and then rendering using the current context.

Whenever we tell OpenGL that we now want to draw lines instead of triangles for example, we change the state of OpenGL by changing some context variable that sets how OpenGL should draw. As soon as we change the context by telling OpenGL it should draw lines, the next drawing commands will now draw lines instead of triangles.

When working in OpenGL we will come across several state-changing functions that change the context and several state-using functions that perform some operations based on the current state of OpenGL. As long as it is kept in mind that

OpenGL is basically one large state machine, most of its functionality will make more sense.

Graphics 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 takes as input a set of 3D coordinates and transforms these to colored 2D pixels on your screen. 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 and can easily be executed in parallel. Because of their parallel nature, graphics cards of today have thousands of small processing cores to quickly process your data within the graphics pipeline. The processing cores run small programs on the GPU for each step of the pipeline. These small programs are called shaders.

Some of these shaders are configured by the developer which allows us to write our own shaders to replace the existing default shaders. This gives us much more fine-grained control over specific parts of the pipeline and because they run on the GPU, they can also save valuable CPU time. Shaders are written in the OpenGL Shading Language (GLSL).

The different parts of graphics pipelining are described below in brief:

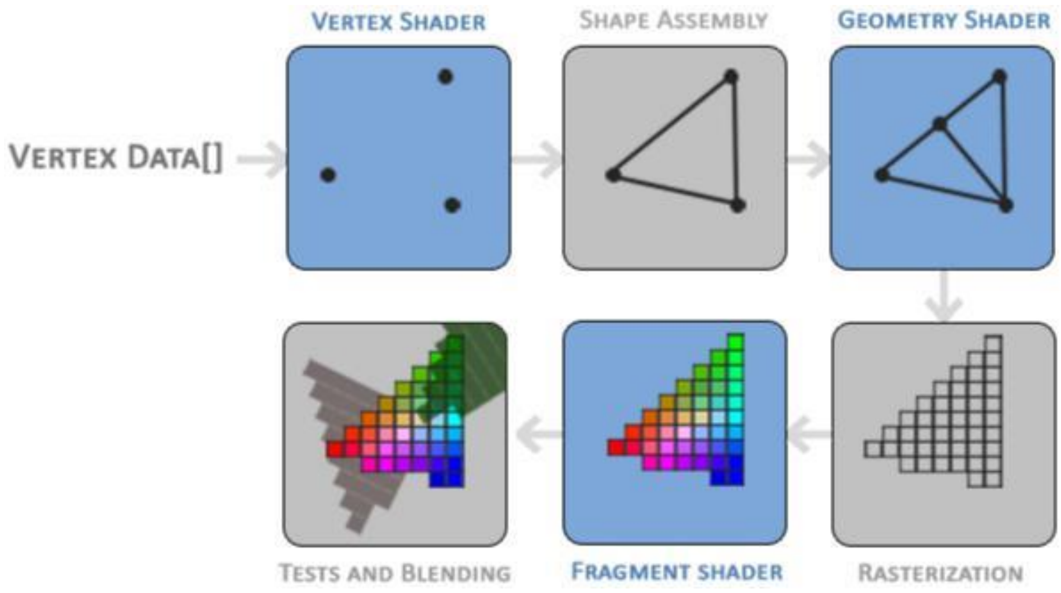


Figure: Different Stages of Graphics Pipelining

Vertex Shader

The first part of the pipeline is the vertex shader that takes as input a single vertex. The main purpose of the vertex shader is to transform 3D coordinates into different 3D coordinates like world coordinates and viewing coordinates and the vertex shader allows us to do some basic processing on the vertex attributes.

Primitive Assembly

The primitive assembly stage takes as input all the vertices from the vertex shader that form a primitive and assembles all the points in the primitive shape given; in this case a triangle.

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 new (or other) primitive(s).

Rasterization Stage

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

A fragment in OpenGL is all the data required for OpenGL to render a single pixel.
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. Usually the fragment shader contains data about the 3D scene that it can use to calculate the final pixel color (like lights, shadows, color of the light and so on).

Alpha Test and Blending Stage

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.

Model Loading

A very popular model importing library out there is called Assimp that stands for Open Asset Import Library. Assimp is able to import dozens of different model file formats (and export to some as well) by loading all the model's data into

Assimp's generalized data structures. Because the data structure of Assimp stays the same, regardless of the type of file format imported, it abstracts all the different file formats out there.

When importing a model via Assimp it loads the entire model into a scene object that contains all the data of the imported model/scene. 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 (simplistic) model of Assimp's structure is shown below:

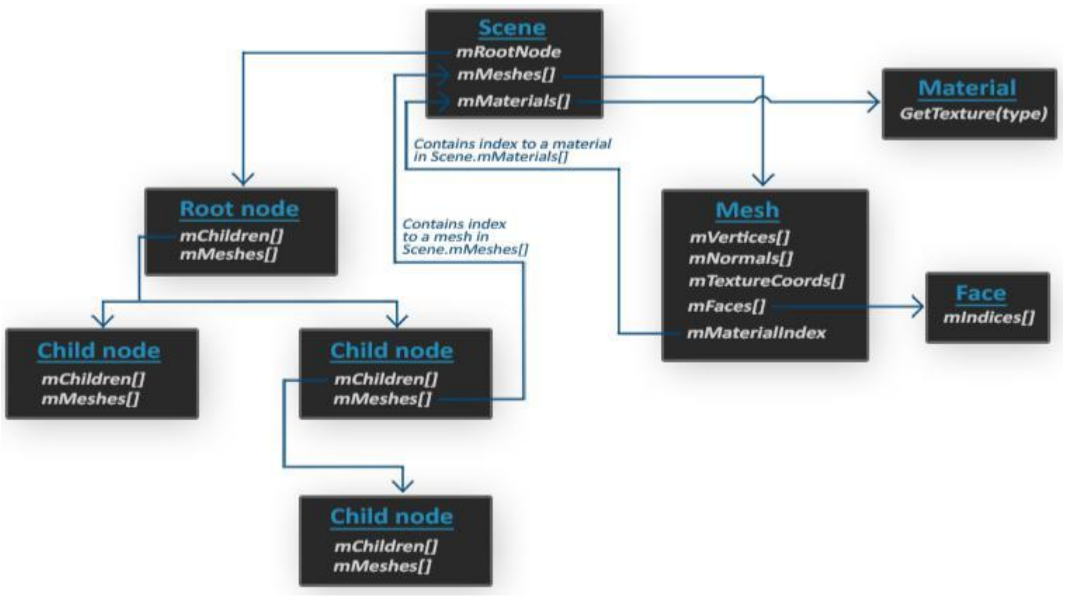


Figure: Simplistic structure of Assimp after model is loaded

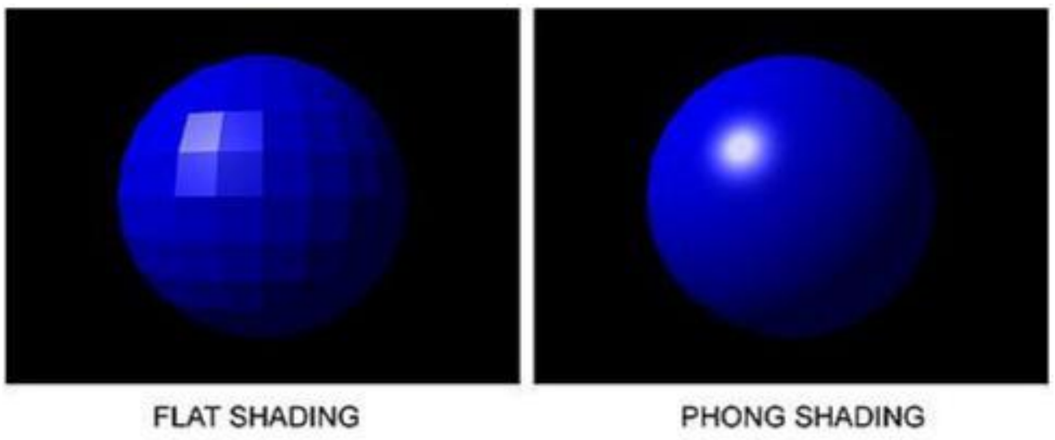
- All the data of the scene/model is contained in the Scene object like all the materials and the meshes. It also contains a reference to the root node of the scene.
- The Root node of the scene may contain children nodes (like all other nodes) and could have a set of indices that point to mesh data in the scene object's mMeshes array. The root node's mMeshes array contains the actual Mesh objects, the values in the mMeshes array of a node are only indices for the scene's meshes array.
- A Mesh object itself contains all the relevant data required for rendering, think of vertex positions, normal vectors, texture coordinates, faces and the material of the object.

- A mesh contains several faces. A Face represents a render primitive of the object (triangles, squares, points). A face contains the indices of the vertices that form a primitive. Because the vertices and the indices are separated, this makes it easy for us to render via an index buffer.
- Finally a mesh also contains a Material object that hosts several functions to retrieve the material properties of an object.

Phong Shading

Phong shading is an interpolation technique for surface shading in 3D computer graphics. It interpolates surface normals across rasterized polygons and computes pixel colors based on the interpolated normals and a reflection model. The Phong model reflects light in terms of a diffuse and specular component together with an ambient term. The intensity of a point on a surface is taken to be the linear combination of these three components.

Figure: difference between flat and phone shading



CubeMaps

A cubemap is a texture that contains 6 individual 2D textures that each form one side of a cube: a textured cube. Why bother combining 6 individual textures into a single entity instead of just using 6 individual textures? Well, cube maps have the useful property that they can be indexed/sampled using a direction vector. Imagine we have a 1x1x1 unit cube with the origin of a direction vector residing at its center. Sampling a texture value from the cube map with an orange direction vector looks a bit like this:

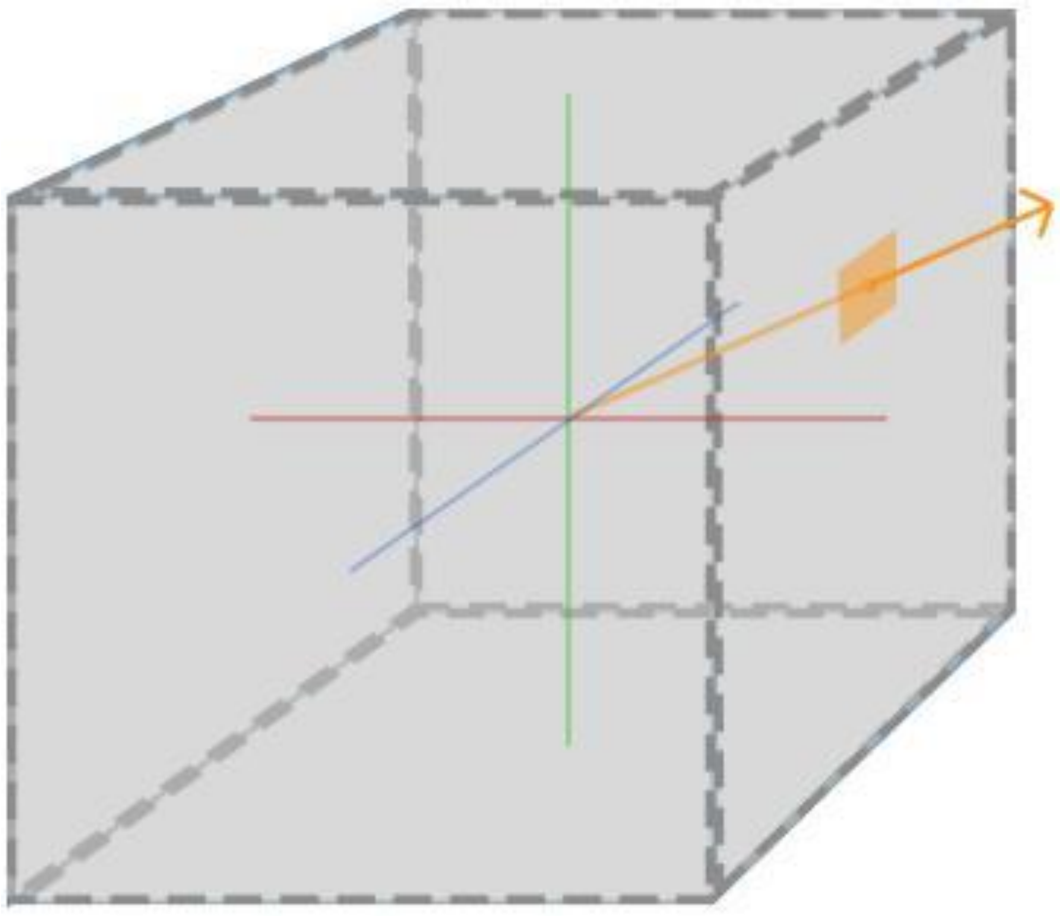


Figure: Cubemap Selecting a TexCoord using Direction Vector

Skybox

A skybox is a (large) cube that encompasses the entire scene and contains 6 images of a surrounding environment, giving the player the illusion that the environment he's in is actually much larger than it actually is. Some examples of skyboxes used in video games are images of mountains, of clouds, or of a starry night sky.

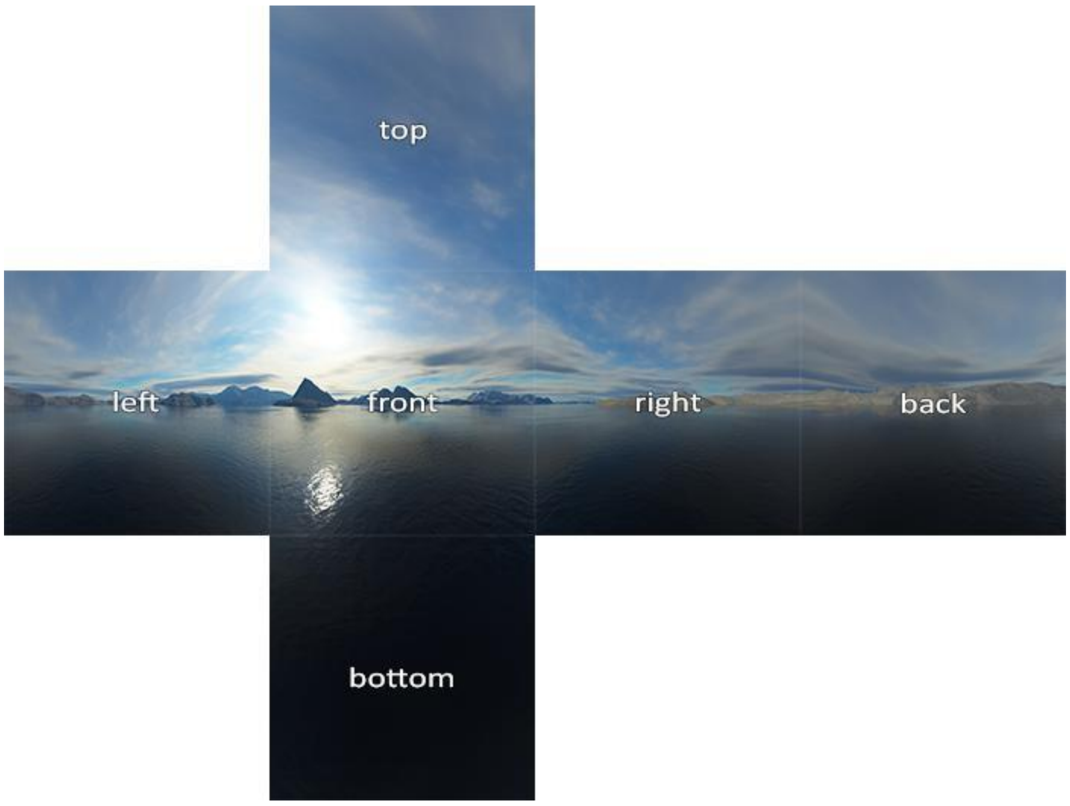


Figure: Skybox and its face in cubemap used in our project

A cubemap used to texture a 3D cube can be sampled using the local positions of the cube as its texture coordinates. When a cube is centered on the origin (0,0,0) each of its position vectors is also a direction vector from the origin. This direction vector is exactly what is needed to get the corresponding texture value at that specific cube's position. For this reason position vectors are only supplied and texture coordinates are not needed.

The skybox is centered around the player so that no matter how far the player moves, the skybox won't get any closer, giving the impression the surrounding environment is extremely large. We remove the translation part of the view matrix so only rotation will affect the skybox's position vectors.

Literature Survey

Since this project is based on the application of concepts of Graphics Programming using OpenGL, different books were suggested to refer to by seniors and experienced programmers for the knowledge of how OpenGL works and the implementation of those Graphics concepts in terms of computer programs. Tutorial websites like learnopengl.com referred to as reference for the development of programs that assisted us to clear the concepts regarding the graphics programming and make our graphics development easier. Additionally, videos of Yan Chernikov a.k.a. The Cherno was viewed on YouTube for the reviewing of concepts and providing an abstraction to OpenGL codes along with the insights to how graphics programming is done in gaming industries like *EA Sports*. Docs.gl website proved to be of great use as documentation for providing the information related to the arguments required for different OpenGL functions along with their return parameters and their work.

Methodology

In the initial phase of development of this project, the fundamental concepts required to proceed in writing the programs were studied. The basics of 3D Computer Graphics like the graphics pipeline, 3D transformations, 3D surface modelling and different illumination models were thoroughly studied. After being familiar with the notion of 3D computer graphics, the essential fields for the accomplishment of the project were discussed. Then the project was divided into three modules:

1. Model Design
2. Model Loading
3. Lighting

The model for the Dining Room was prepared using the Blender software. Different reference pictures were consulted during the creation of a 3D model of the building. After the completion of the model designing, the necessary programs required to load the model were written by using OpenGL in C++. To accomplish the task of model loading, an open source model loader namely *Assimp* was used. After successfully loading the model, camera movement was also added to the project. Next, programs to achieve lighting effects were also written and tested. To perform various mathematical calculations in the programs written for the project, a mathematics library *OpenGL Mathematics(GLM)* was used extensively. Upon the completion of writing all the programs and integrating them together into a single module, appropriate testing was performed to make sure the program worked as expected. Finally after undergoing various phases of development, we were able to write a 3D model for the Dining Room.

Implementation

Block Diagram

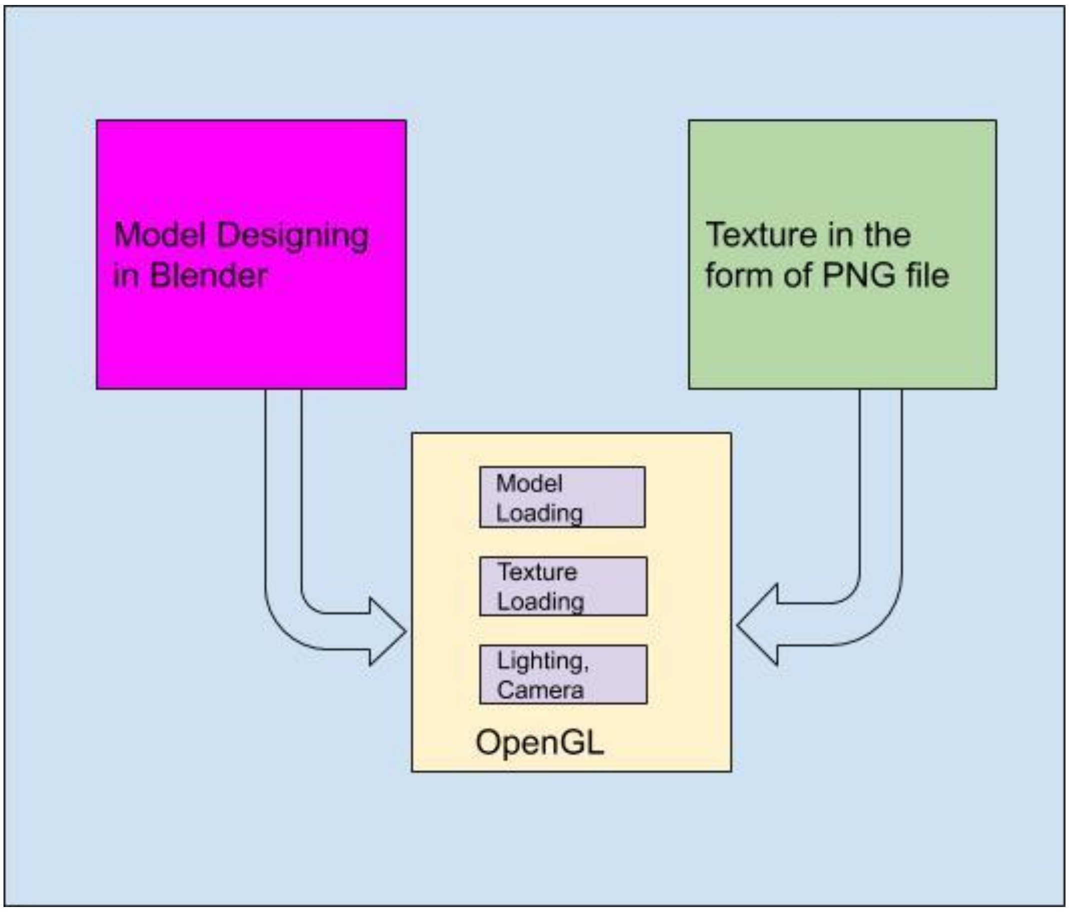


Figure: Block Diagram for Overview of Project

Results

The Dining Room was modelled in Blender and exported in *.fbx* format along with diffusion and specular maps. The exported files were imported from our rasterizer to see the objects from different views along with the effect of light on the objects. Lighting was done using the *Phong's model* to make lighting more appealing.

Problems Faced and Solutions

The main problem faced during this project is the unfamiliarity with GPU pipelining. The whole GPU pipelining concept of *OpenGL* was completely new to us. Moreover, our course of *Computer Graphics* also doesn't focus on GPU pipelining. Learning OpenGL after learning the steps and procedures of GPU Pipeline is easy but learning OpenGL along with it is not an easy task. So we had to go through a complete new book from learnopengl.com to learn all those new concepts.

Another problem faced was learning Blender along with the programming stuff. One of our team members had to dedicate his time to *Blender*, which led to giving less time in OpenGL and graphics programming. This occurred because we didn't have a choice but to model Dining Room yourself because the design of Dining Room was what we have been using in our daily life and said its Dining Room.

Another problem faced was in loading the exported model file from *Blender* to our own rasterizing engine. Problems were faced in exporting the file from Blender to the current format needed for our system to load. Moreover, there occured a bug in loading *.obj* files using the third party library we have been using for loading model, *Assimp*. So, we had to switch our model format from *.obj* to *.fbx*.

In addition, we faced the problem of texture baking of the material of the blender model. The texture file of the table, chairs, floor, scene etc. were exported separately.

Limitations and Future Enhancements

As we've completed the project in a limited amount of time, many more features can be added to make the project more attractive. The following features can be added to our project:

1. The current system supports only a maximum of 2 diffusion and 2 specular maps for loading coefficients of diffusion and specular lighting from textures.
2. The current system doesn't show a walking person as in the Third Person Shooter game and the corresponding camera movement.
3. The collision checking of the camera and model was not completed in the current model due to time constraints.
4. The accurate modelling of Dining Room with all the interiors was not viable in this short span of time so only the table, chairs, window, floor of Dining Room was modelled in detail from *Blender*.
5. The accurate lighting maps and model were not found and, any more time couldn't have been invested in modeling any further model.

Conclusion and Recommendations

This project was unquestionably a good way of learning and implementing the way for graphics programming. This project leads us to the winding up of the graphics programming that for developing graphics modelling and rendering software a good judgment and proper analysis of the topic is required at first rather than the coding. Coding is not the initial step for emergence of any program, rather a good planning on the basic framework and making decisions on the way of implementing the program is the best. After the coding of the program the system may not be as per our requirement but debugging, if any error, and testing and execution of the program are furthermore required. After the completion of the system, its management takes another most required obsession that is to be handled with great care.

Thus, after the completion of our project, we can conclude proper judgment and implementation of the problems or topic leads to good programming practice which might lead to having desired output.

Appendix

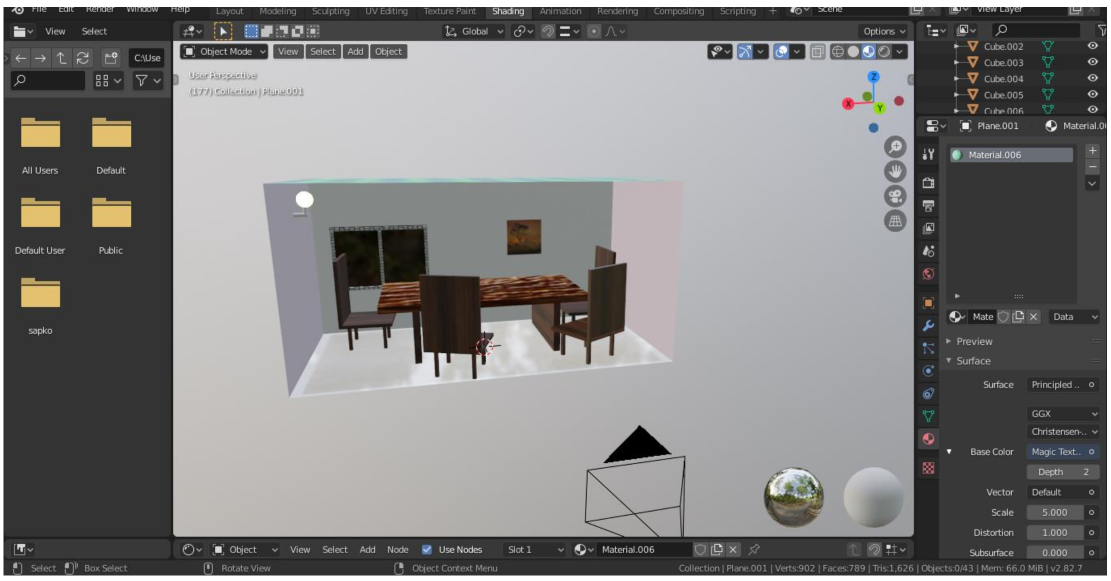


Figure: Model designing in Blender

Figure: Dining Room Model Simulation

References

1. Yan Chernikov, OpenGL,YouTube
2. Joey DeVries, learnopengl.com
3. Sean Barrett, nothings.org
4. G-Truc Creation, OpenGL Mathematics (GLM)
5. Jorge Rodríguez,docs.gl