TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A

PROJECT

REPORT ON

3D Modeling of Vatsala Devi

SUBMITTED BY:

Bigyapti Bashyal (076BCT016)

Bivek Shrestha (076BCT020)

Ishani Malla (076BCT028)

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

3rd September, 2022

Acknowledgement

We would like to thank the Department of Electronics and Computer Engineering , Pulchowk Campus who gave us the opportunity to implement theoretical knowledge practically through this project.We had a wonderful chance to apply the algorithms and methods we learned about in computer graphics into practice. We were able to learn more about how the graphics we use on a regular basis function.

Our sincere thanks to our Lecturer , Basanta Joshi, without whose guidance this project would not have been possible. We would also like to express gratitude towards the entire Lab department for their support.It helped us to gain insight to the working of graphics with which we interact in our daily life. It is a great pleasure to understand how the inner things work with different algorithms involved that we are unaware of when using it.

We further extend this thanks to our family members and friends who have encouraged and assisted us through every step of the way.

Abstract

This project utilizes computer graphics to generate a 3D model of the vatsala devi temple. Computer graphics involves the use of computers to create and manipulate pictures on a display device. It comprises software techniques to create, store, modify, and represent pictures. Computer graphics techniques that increase the richness of a visual rendering include advanced shading methods, lighting, texture mapping, translucency, and atmospheric effects. 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 Content

Introduction

The Vatsala Durga Temple is located Bhaktapur Durbar Square just in front of palace and beside king's monument.This is Shikhara-style temple is entirely made of sandstone and stand on three stage plinth,comparable to Patan's Krishana Mandir.Vatsala Devi, a form of goddess Durga, is honored here.

In 1696A.D. King Jitamitra Malla constructed the temple.However, the current edifice was built by King Bhupitindra Malla and originates from the late 17th or early 18th century.

Description

Our project titled "3D modeling of Vatsala Devi" is the simulation model of the Vatsala Devi Temple.The landscape features a temple and its surroundings . Users have the ability to navigate the camera in the 3D model by using the keys: A, W, S, D and the mouse scroll for zoom in and zoom out.

Objectives

The major objectives of our project are:

1. To design and render a beautiful 3D model of Vatsala Devi temple.
2. To be familiar with blender for making 3D models.
3. To learn about OpenGl for rendering 3D objects.
4. To implement the graphics algorithms to create real life objects.
5. To understand the concept of shaders and lighting models.
6. To work with the team members in mutual collaboration.

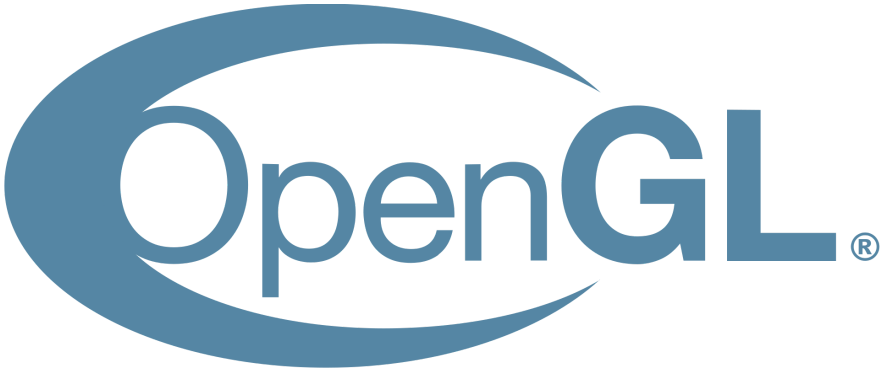
Background Theories

1.**Blender**:

Blender is a free and open-source 3D computer graphics software tool set used for creating animated films, visual effects, art, 3D-printed models, motion graphics, interactive 3D applications, virtual reality, and, formerly, video games. It is well suited to individuals and small studios who benefit from its unified pipeline and responsive development process. Being a cross-platform application, Blender runs on Linux, macOS, as well as Windows systems. It also has relatively small memory and drive requirements compared to other 3D creation suites. Its interface uses OpenGL to provide a consistent experience across all supported hardware and platforms.

2. **OpenGl**

OpenGL (Open Graphics Library) is a cross-platform, hardware-accelerated, language-independent, industrial standard API for producing 3D (including 2D) graphics. Modern computers have a dedicated GPU (Graphics Processing Unit) with its own memory to speed up graphics rendering. OpenGL is the software interface to graphics hardware. In other words, OpenGL graphic rendering commands issued by your applications could be directed to the graphic hardware and accelerated.



3. **GLFW**

GLFW is an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating windows, contexts and surfaces, receiving input and events. GLFW is written in C and supports Windows, macOS, X11 and Wayland. GLFW is licensed under the zlib/libpng license.

It allows us to create an OpenGL context, define window parameters, and handle user input, which is plenty enough for our purposes.

4.**GLM**

OpenGL Mathematics (GLM) is **a C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specification**.

5. **GLAD**

Glad is a multi-language loader generator for the Khronos Group's official GL/GLES/EGL/GLX/WGL specifications. It quickly adapts to various languages and provides a loader for the developer's requirements based on the most recent specifications. The GLAD libraries are available at https://glad.dav1d.de/.

6. **GLSL**

The primary shading language for OpenGL is the OpenGL Shading Language (GLSL). While numerous shading languages can be used with OpenGL thanks to OpenGL Extensions, GLSL are directly supported by OpenGL without extensions.
GLSL is a C-style language. The language inherits OpenGL's deprecation approach and has experienced a number of revisions. The most recent GLSL release is 4.60.

7. **Shader**

A Shader is a user-defined program designed to run on some stage of a graphics processor. Shaders provide the code for certain programmable stages of the rendering pipeline. They can also be used in a slightly more limited form for general, on-GPU computation.There are mainly two types of shaders

Vertex Shaders

The most well-known and often used type of 3D shader, vertex shaders, are executed once for each vertex provided to the graphics processor. The objective is to convert each vertex's 3D position in virtual space to its 2D coordinate in screen space (as well as a depth value for the Z-buffer). Although they can alter a vertex's position, color, and

texture coordinates, vertex shaders cannot add additional vertices to an object. The output of the vertex shader is passed to the following stage in the pipeline. Any scenario incorporating 3D models can benefit from tremendous control over the specifics of position, movement, lighting, and color thanks to vertex shaders.

Fragment Shader

The Shader stage, known as a "Fragment Shader" , transforms a rasterized fragment into a palette of colors and a single depth value.
After a primitive has been rasterized in the OpenGL pipeline, the fragment shader is the next stage. A "fragment" is created for each sample of the pixels a primitive covers. Each fragment contains all of the interpolated per-vertex output values from the previous Vertex Processing stage along with a Window Space position and a few other values.
A fragment shader produces a depth value, a potential stencil value (which is unaltered by the fragment shader), and zero or more color values that could be transferred to the framebuffers as of right now.

8.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.

9. 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.

10. Coordinate Systems in Computer Graphics

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.

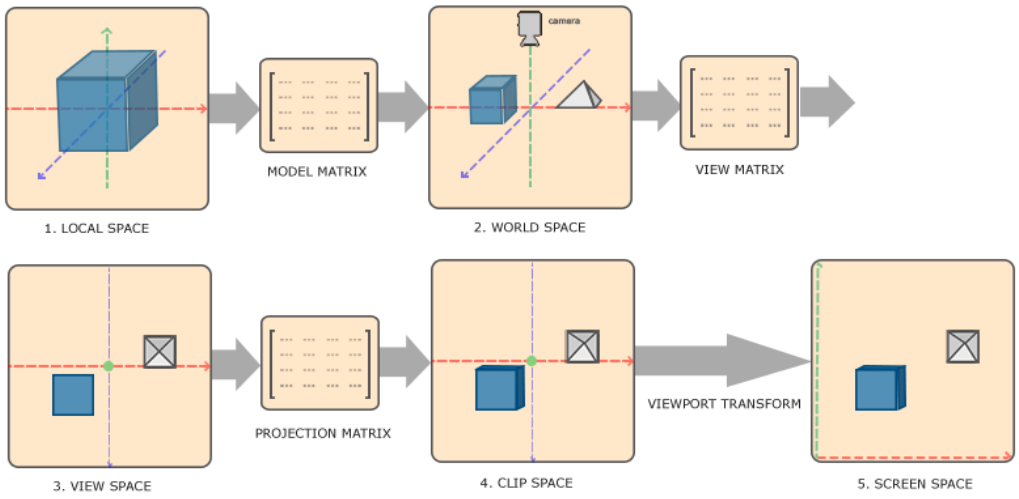
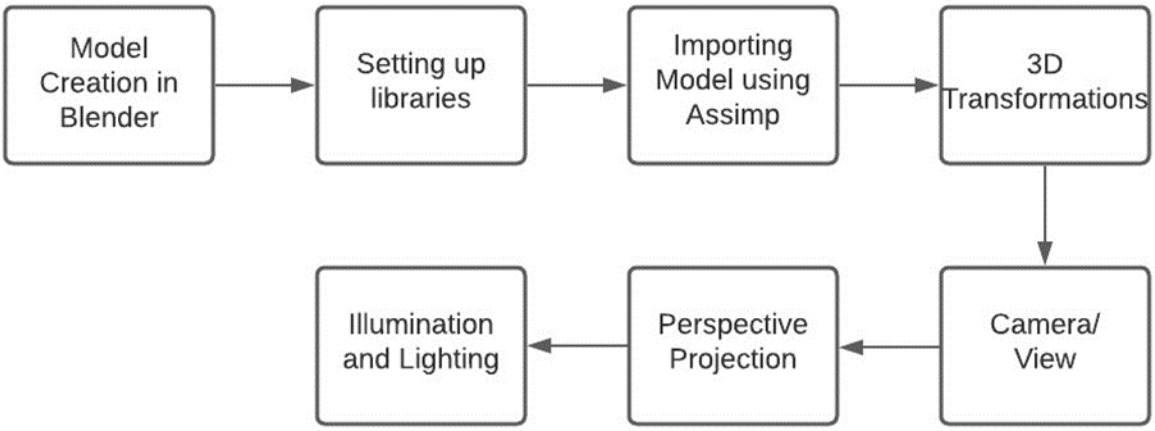


Fig : Graphics Pipeline

Methodology



Blender Model

The models of Vatsala Devi were initially made in Blender utilizing various mesh, curves, and functionalities. The model was given various textures and colors before being exported as an obj file. The model's vertices, texture coordinates, normals, and faces were all included in this created obj file as triangulated meshes. Blender also produced a material file (mtl) that linked the different attributes of materials.

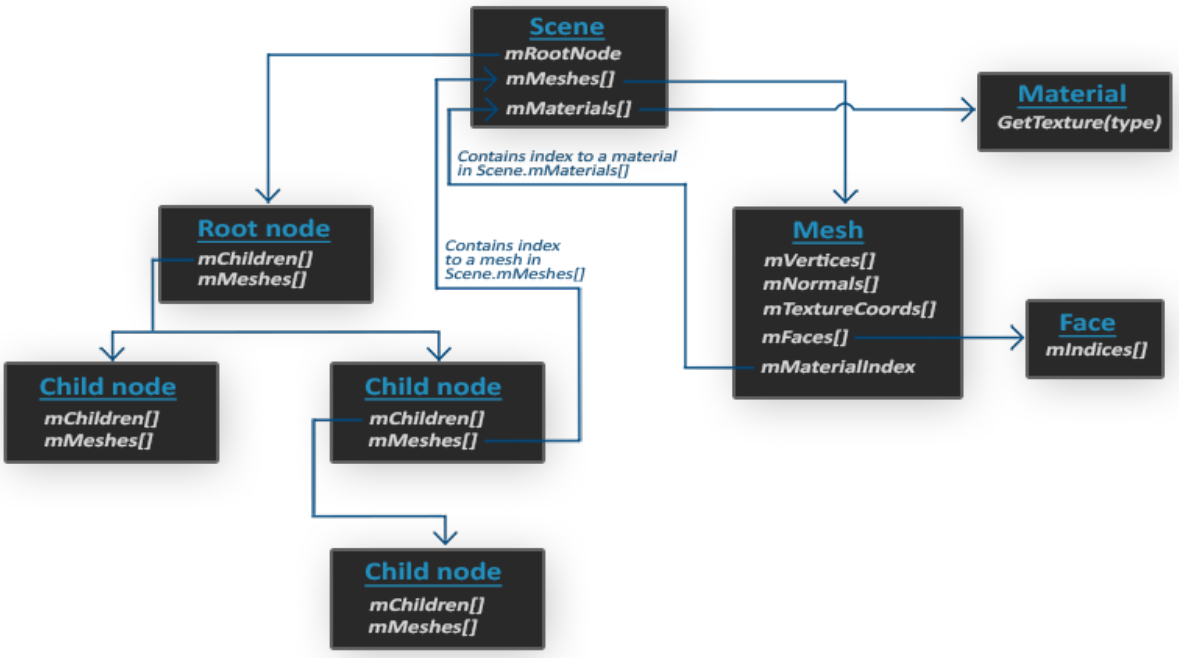
Setting Up Libraries

Our project uses a variety of libraries to carry out particular functions. They are as follows:

1. GLFW: responsible for creating and managing windows and processing user inputs.
2. GLAD: used to search and load OpenGL add-ons.
3. glm:OpenGL Mathematics (GLM) is a header-only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications.
4. Assimp: a library developed for easy loading and rendering of standard 3D models.
5. stb_image: utilized to load various textures used in the 3D model.

Importing Model using assimp

The model made in Blender was introduced to the code using the Assimp Library. The Assimp (Open Asset Import Library) parses these exported model files and extracts all the relevant information so we can store them in a format that OpenGL understands.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:



Mesh Configuration

The Mesh loaded from OpenGl is conditioned to reflect what the code understands. In this process the mesh is converted to an array of vertices and attributes which is used in the next step during 3D Transformation.

3D Transformation

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 the 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.

Camera

Camera is the port through which the user views the world. So, to obtain a realistic view of the object, the camera can be moved using keyboard and mouse to view the object from different directions. This is done by translating the camera position, rotating the camera field of view.

Perspective Projection

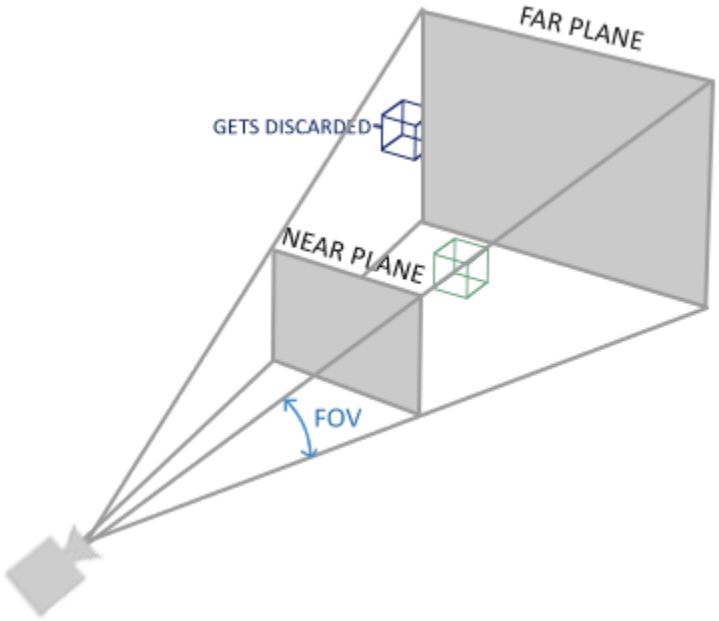
The projection matrix maps a given frustum range to clip space, but also manipulates the w value of each vertex coordinate in such a way that the further away a vertex coordinate is from the viewer, the higher this w component becomes. Each component of the vertex coordinate is divided by its w component giving smaller vertex coordinates the further away a vertex is from the viewer.

A perspective projection matrix can be created in GLM as follows:

```
glm::mat4 proj = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f, 100.0f);
```

Its first parameter defines the fov value that stands for field of view and sets how large the viewspace is. For a realistic view it is usually set to 45 degrees, but for more doom-style results you could set it to a higher value. The second parameter sets the aspect ratio which is calculated by dividing the viewport's width by its height. The third and fourth parameters set the near and far plane of the frustum.

What glm::perspective does is creates 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. An image of a perspective frustum is seen below:



This projection is applied via a projection matrix created by glm as shown below:

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

Depth-Buffer Method (Z-Buffer)

A float array is generally used as the z buffer; each pixel's z buffer is stored in the array. This array is used as a decision parameter that helps to decide if the currently existing pixel can be

overdrawn or not, if it needs to be overdrawn, the z buffer is updated along with the pixel array, else it is ignored. In OpenGL, this can be done by calling:

```
glEnable(GL_DEPTH_TEST);
```

And each frame, the z buffer can be cleared by calling:

```
glClear(GL_DEPTH_BUFFER_BIT);
```

Clipping

Each triangle in the scene is clipped once against the near plane (before projection) and finally against the four sides of the screen before rendering. It is simply done by interpolating to calculate the clip point which was used to divide the triangle into smaller clipped triangles. This is automatically done by OpenGL.

Phong Illumination

Though we use flat shading, the computations done were using the phong illumination model.

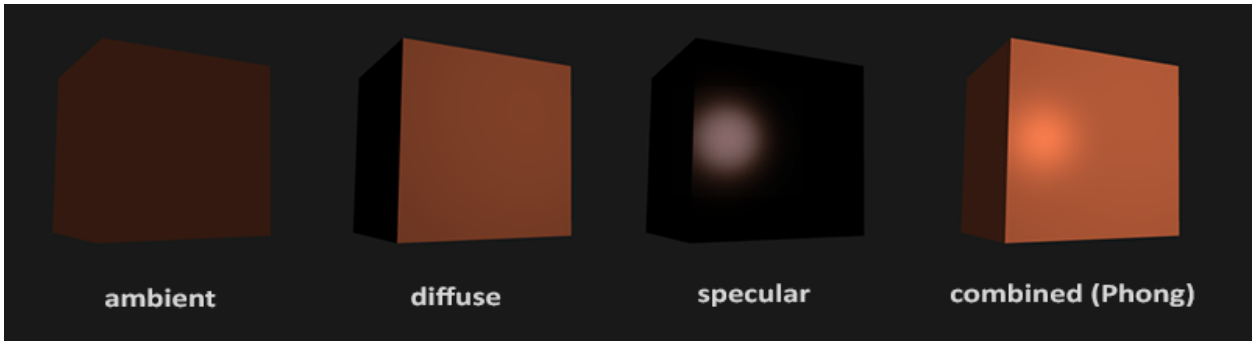


Fig: Phong Illumination Model

$$f(\vec{p}) = \underbrace{I_a K_a}_{\text{ambient component}} + \underbrace{\sum_{i}^{\text{nb\_lights}} (\vec{n}(\vec{p}).\vec{l}_i) K_d I_i}_{\text{diffuse component}} + \underbrace{\sum_{i}^{\text{nb\_lights}} f_{spec}(\vec{l}_i(\vec{p}), \vec{v}(\vec{p})) K_s I_i}_{\text{specular component}}$$

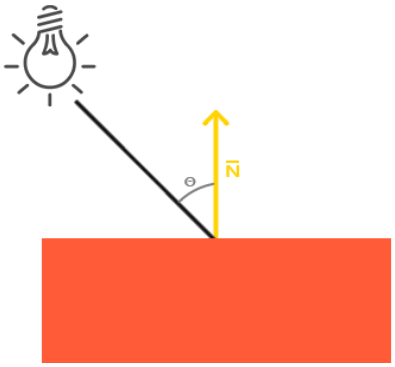
Ambient component

In phong lighting we don't simulate light bouncing in the scene, but if we don't take this phenomenon into account, parts of the object not directly exposed to the source light would stay black. To fix this, the ambient component arbitrarily assigned a fixed intensity. This intensity only depends on the material of the object (assigned by the user).

- Ka intensity of the material (real number, user defined)
- Ia intensity of the scene (real number, user defined)

Diffuse component

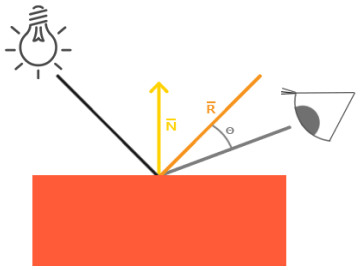
Simulate the light that is scattered uniformly where it strikes the object's surface (e.g. plaster), it only depends on the light direction. The intensity is independent from the viewing point since light is diffused uniformly.



- Kd diffuse intensity of the material (real number, user defined)
- Ii diffuse intensity of the ith light (real number, user defined)
- n normal vector at the position p on the object's surface (3D vector)
- li direction of the light striking at p (e.g. li=lightPosition−p) (3D vector)

Specular component

Simulate light that is reflected off the object, therefore it depends on both the light source position and viewer perspective.



- fspec(li(p),v(p)) specular term described below (real number)
- v viewer direction (e.g. li=cameraPosition−p) (3D vector)
- Ks specular intensity of the material (real number) (user input)

$$\begin{cases} f_{spec} & = & (\mathbf{r}.\mathbf{v})^c & \text{if } \mathbf{r}.\mathbf{v} > 0 \\ f_{spec} & = & 0 & \text{otherwise} \end{cases}$$

Where c is coefficient representing the size of the specular reflection (user input) and $\mathbf{r}$ is the reflection of the light vector $\mathbf{l}$:

$$\mathbf{r} = (2(\mathbf{n}.\mathbf{l})\mathbf{n}) - \mathbf{l}$$

Language and tools

FINAL PROJECT:

Languages used:

- C++
- GLSL

Libraries Used:

- OpenGl
- GLAD
- GLFW
- Assimp
- Blender
- Glm
- stb_image

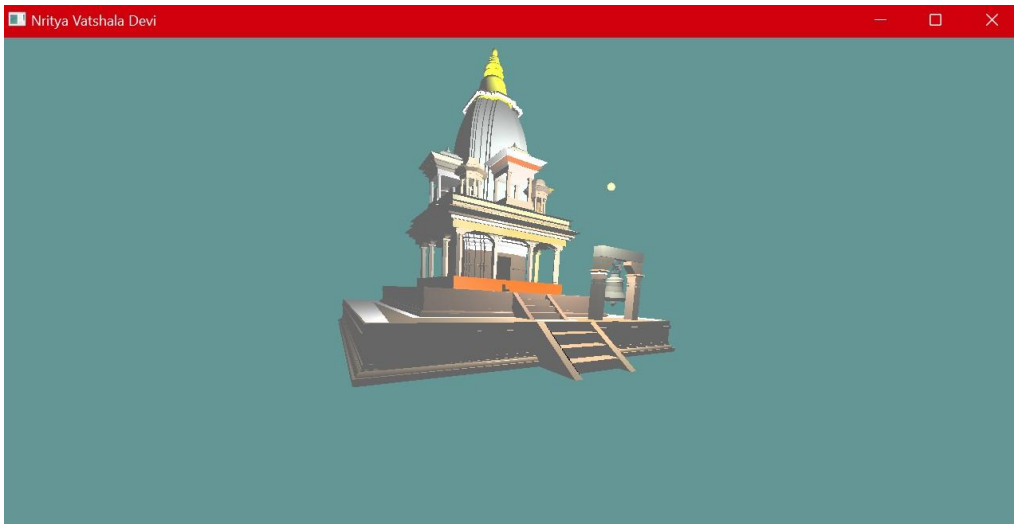
MIDTERM :

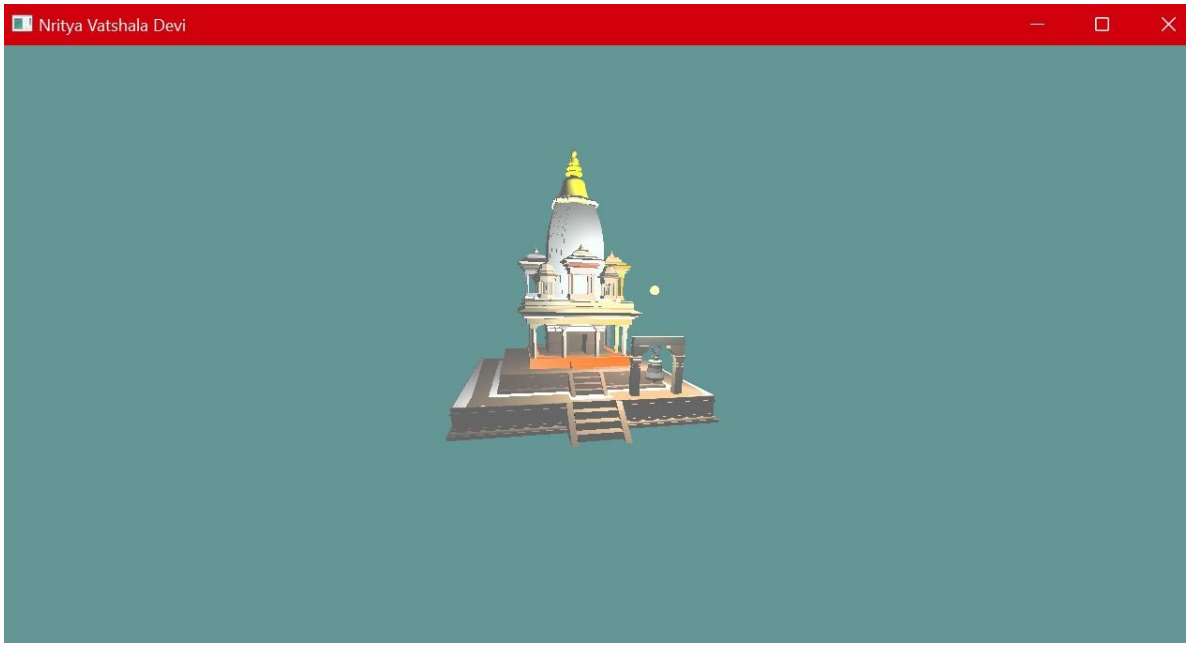
Languages used:

- C++
- SFML

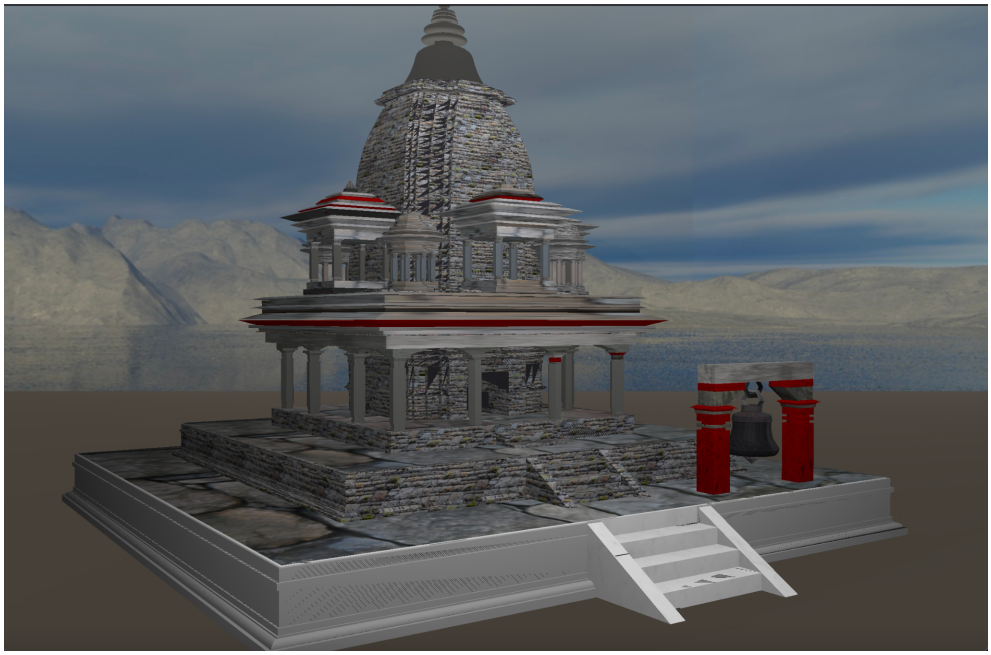
Midterm vs Final :

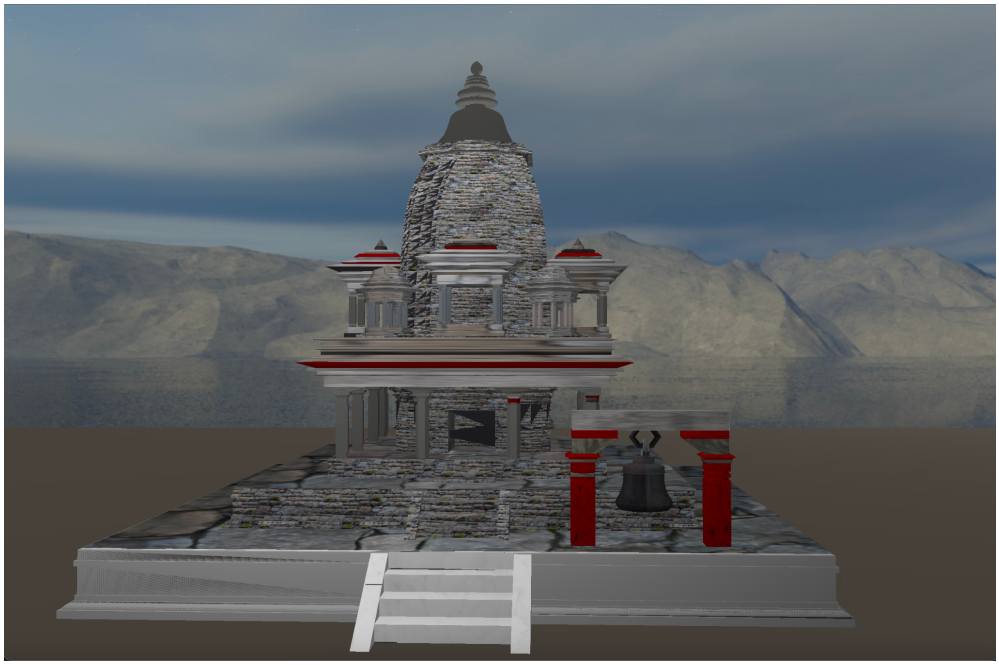
In our midterm project , the blender model was incomplete and did not have textures. We were only able to render at an average frame rate of 15fps. We did not implement clipping. We only added a single point source of light.





In our final project, we improvised on the model and added textures. We were able to render at higher frame rates i.e. around 60fps. We added delta time calculations (the time between the current frame and the next frame) to ensure uniformity while rendering among different computers that have different processing speed. We implemented clipping. We added 2 point sources of light.





Major improvements in final project

Better Shading

The midterm used flat shading even though a phong illumination model was used.Approximations were made throughout the surface making it unrealistic. This was fixed in the final version by using shaders which provided a per pixel computation platform using fragment shaders.

More realism

The midterm didn't have any realistic coloring as well. The midterm project which did not support HDR colors. Final build through opengl supports HDR colors making it more realistic.

Model Loading:

We wrote a model loader in the midterm project. This was done by parsing the wavefront obj file and simply filling the triangle mesh structs we made. In the final ASSIMP is used for model loading.

Midterm Problems

Since the midterm project was done utilizing only the CPU , many things including phong shading which was something we couldn't implement in our project due to the lack of parallel processing via the GPU.

Final Problems

Improvised model caused a drop in fps though it was much faster in comparison to midterm. Debugging glsl was difficult.

Conclusion

We were able to render a 3D model of Vatsala Devi using this method. We used the algorithms that we learned in our course with success.

Limitation

- The rendered scene is less realistic.
- Textures look pixelated on zooming in
- Lags might be observed between the user input and camera movement in the scene.
- The light source illuminates only the temple making it less realistic
- Cannot switch between different views of the model without camera movement

References

https://www.blender.org/

Scratchapixel

learnopengl.com