



TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

A
REPORT ON
ROCKET MODELING

Submitted By:

Pranjal Pokharel(075BCT061)

Tilak Chad(075BCT094)

Udeshya Dhungana(075BCT095)

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

LALITPUR, NEPAL



TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

A REPORT ON
ROCKET MODELING

Submitted to :

Asst. Prof Anil Verma

By:

Pranjal Pokharel(075BCT061)

Tilak Chad(075BCT094)

Udeshya Dhungana(075BCT095)

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
LALITPUR, NEPAL

Acknowledgment

We would like to express our deepest gratitude to the Department of Electronics and Computer Engineering, IOE, Central Campus, Pulchowk for providing us an opportunity to work on a graphics project, which is likely to be very helpful for shaping our knowledge and career.

We would also like to express our deepest thanks to our subject teacher Mr. Anil Verma for his suggestions during the project development. We would also like to express our gratitude to our classmates who were there in discussion whenever we needed it. We acknowledge the support of our family, without whom all these were impossible.

Last but not least we would like to thank everyone who is involved directly and indirectly in this project.

Abstract

With the rise of processing and memory power of computers, 3D modeling is not a new thing in the world. In our project “Rocket Modeling” we invested our effort to visualize the overall structure and launchpad of “Vostok - 1”, the rocket in which Yuri Gagarin became the first human to reach space. Blender is used to design the different structures and objects, C++ as a programming language and OpenGL in order to implement graphics in the project. The project consists of proper visualization of Vostok-1 and its launchpad. The rocket is located in the launchpad in a ready to launch position. Through the use of transformation, matrices and input from the keyboard and mouse, the user can move around the entire area and zoom in/out to certain locations. We’ve also added lighting effects to the entire area based on the individual fragment’s position with respect to light’s position which gives a real world effect to each object.

Table of Contents

Acknowledgment	3
Abstract	4
Table of Contents	5
List of Figures	7
List of Symbols and Abbreviations	8
Introduction	9
Background	9
Objectives	10
Motivations	10
Scope	10
Literature Review	11
Related Theories	11
Projection	11
Matrix Transformation	12
Phong Shading	14
Z Buffer	15
Related Works	15
Methodologies	16
Graphical Representation of Project Workflow	16
Algorithms and Implementations	17
Projection (Perspective)	17
Matrix Transformations	17
Phong Lighting	18
Z-buffering	19

Camera	20
Languages and Tools	21
Programming Language - C/C++,	21
Build Tool - CMake	21
OpenGL	22
GLFW	22
Blender + Assimp (3D Modeling)	22
Output and Result	22
Conclusion	23
Limitations and Future Works	23
References	24

List of Figures

Vostok Rocket

Workflow Graph

Fig. (a) Parallel Projection (b) Perspective Projection

Viewing volume and viewing plane

Phong Reflection

List of Symbols and Abbreviations

OpenGL -> Open Graphics Library

GLFW -> Graphics Library Framework

2D / 3D -> Two dimensional / Three dimensional

VCS -> Version Control System

Introduction

Background

This is the final report of our Computer Graphics project as a part of 5th semester study of Bachelor in Computer Engineering (BCT) course, Pulchowk Campus, IOE. The Computer Graphics syllabus covers different concepts related to graphics rendering and rasterization from beginner to advanced. As a part of this project, we will model the **Vostok Rocket** as we detailed in our proposal. Between submitting our project proposal on 29th June, 2021 and final project submission date on 27th August, 2021, we had about two months to complete the rendering of the rocket model using various algorithms learned as a part of the course and different open-source projects and libraries available at our disposal.



Fig. Vostok Rocket

Objectives

Our objectives for this project have been discussed previously as a part of our proposal. Nonetheless, we list out a few of them here with some revisions -

1. To learn 3D rendering through projection onto our 2D screens.
2. To learn about OpenGL specification (specifically version 3.3 core).
3. To learn about the general outline of the Graphics Pipeline - this includes shaders, viewports, batch rendering using vertex buffers, matrix transformations, view transformations and more.
4. To understand concepts behind lighting such as ambient, diffuse and specular reflections as well as various shading models (especially the phong shading model).
5. To understand the concept behind depth testing and utilization of z-buffer for discarding surfaces that are not visible as a part of the view.
6. To get comfortable with graphical debugging and loading texture/models for render.
7. To learn Blender 3D for designing professional 3D models and animation.
8. To learn to work in a team using Version Control System (VCS).

Motivations

Our main motivation behind the project is to implement the various graphics algorithms and rendering techniques we learned as a part of our curriculum. This includes curve and polygon rendering, two and three dimension transformations, surface modeling, visual surface detection and illumination. Our project tries best to implement these concepts with as much accuracy, simplicity and clarity as possible.

Scope

The scope of our project can be detailed in the following points -

1. 3D models can be loaded as a part of C++ code which allows us for more flexibility in model presentation and manipulation.
2. Lighting system that can be added to any 3D working space.
3. 3D models of rockets can help people in the field of aerospace to better visualize and understand the core concepts they are expected to learn in their curriculum - our project can be extended to have educational value as a teaching material.

Literature Review

Related Theories

1. Projection

There are two most common ways to project our 3D models so that they appear 3D on our 2D screens - parallel and perspective projection. Parallel projection produces unrealistic results since the projection directly maps coordinates into 2D plane and all lines of sight from the object to projection plane are parallel i.e. it doesn't take viewer perspective into account. So, instead we chose to adopt perspective projection.

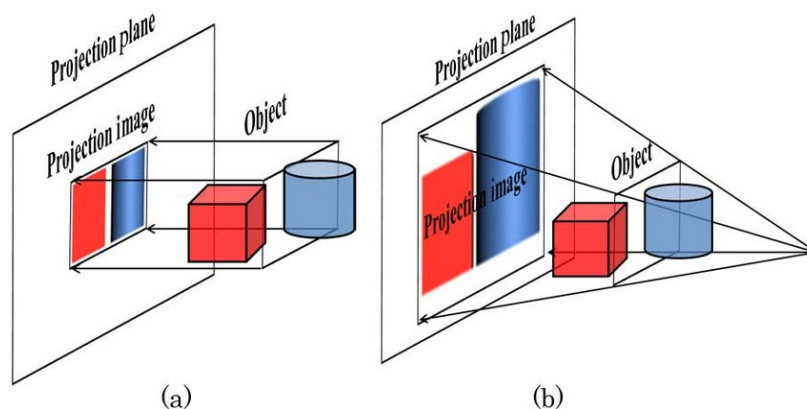


Fig. (a) Parallel projection, (b) Perspective projection

In perspective projection, the distance from the **center of projection (COP)** to the projection plane is finite and so the size of the object varies inversely with distance i.e. distant objects appear smaller than nearer objects. The center of projection is the point at which all lines of sight appear to converge.

To create a perspective view, we first create our projection matrix which we then multiply with the vertex coordinates of the object to transform into perspective projection.

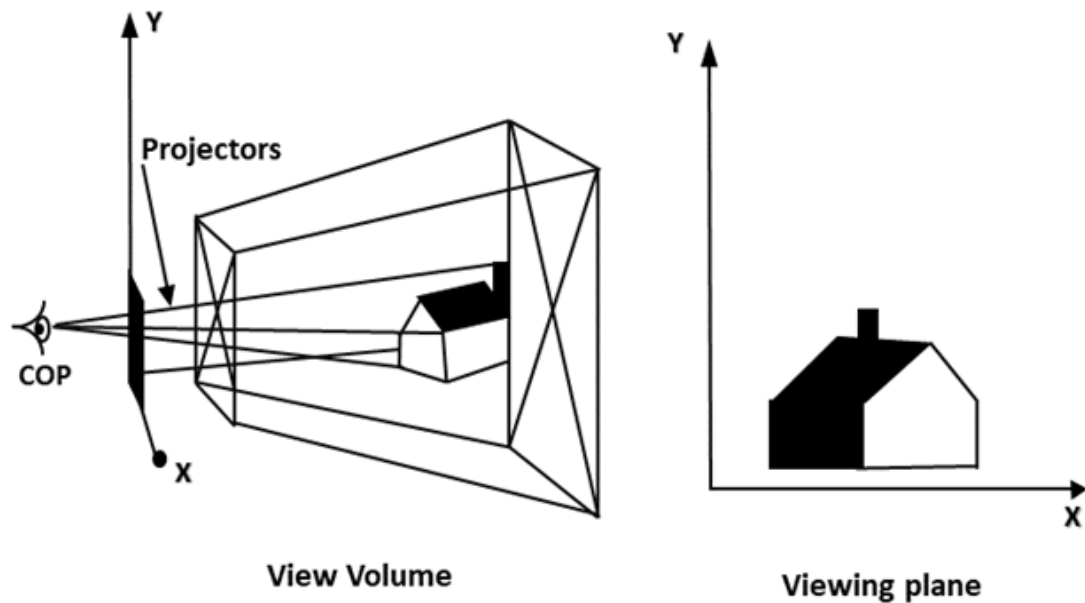


Fig. View Volume and Viewing Plane (Perspective Projection)


2. Matrix Transformation

We use different types of matrices to manipulate the vertices (vectors) through different stages of the graphical pipeline.

Matrices are used to represent and combine various types of transformations. Transformations can be combined together by multiplying two transformation matrices together.

We use the **translation matrix** to translate the object within the world view.

Translation matrix


$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ z + dz \\ 1 \end{bmatrix}$$

Here, the object is translated by the delta vector $[dx, dy, dz]$ where,

```
dx = change in object's x coordinate  
dy = change in object's y coordinate  
dz = change in object's z coordinate
```

We can change the size of the object by using the **scaling matrix**.

$$\begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x * sx \\ y * sy \\ z * sz \\ w \end{bmatrix}$$

The principal diagonal elements contain the scale factors which scale the various components of the vertex coordinates. The w coordinate is used for perspective projection so it is left untouched.

```
sx = scale factor for x-coordinate  
sy = scale factor for y-coordinate  
sz = scale factor for z-coordinate
```

For rotation in 3D space, we have three options - **rotation about x-axis, y-axis or z-axis**. We could also rotate the object about an arbitrary axis in 3D space, however this is much more expensive in terms of floating point calculations so we generally avoid it.

X-axis rotation	Y-axis rotation	Z-axis rotation
$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

The above matrices are used for rotation about angle Θ in the respective axes.

3. Phong Shading

Phong Shading is an interpolation technique for surface shading. Like other shading techniques, phong shading tries to approximate the local behaviour of the light on the object's surface.

Phong shading interpolates normals across all the pixels of the rasterized polygons, and calculates the lighting information per pixel. It is per pixel shading technique. It provides a better reflection model than its other counterparts. Phong lighting composes of three components -

Ambient: The light of the surrounding and does not depend upon the specific light sources. Ambient light is constant for a given scene.

Diffuse: Diffuse component of light is the directional impact of light source on the surface normal. It depends upon the dot product of direction of light and the normal.

Specular: Specular component is the mirror-like reflection of the object. It depends on the dot product of the reflected vector of light along the normal and the view vector.

Total illumination is given by,

$$\text{Total illumination} = \text{ambient} + \text{diffuse} + \text{specular}$$

4. Z Buffer

A **depth buffer**, also known as a z-buffer, is a type of data buffer used in computer graphics to represent depth information of objects in 3D space from a particular perspective. It is an efficient alternative to Painter's Algorithm which is used for similar purposes. Z buffer solves the visibility problem.

Whenever two or more objects occupy the same pixel on the raster image, z buffer is used to decide which object is visible. The depth value of the current pixel is compared to the previously stored pixel, if it is lower than the previous pixel, the current pixel on the color buffer replaces the previous pixel. And if it's not, the old color pixel is retained.

Related Works

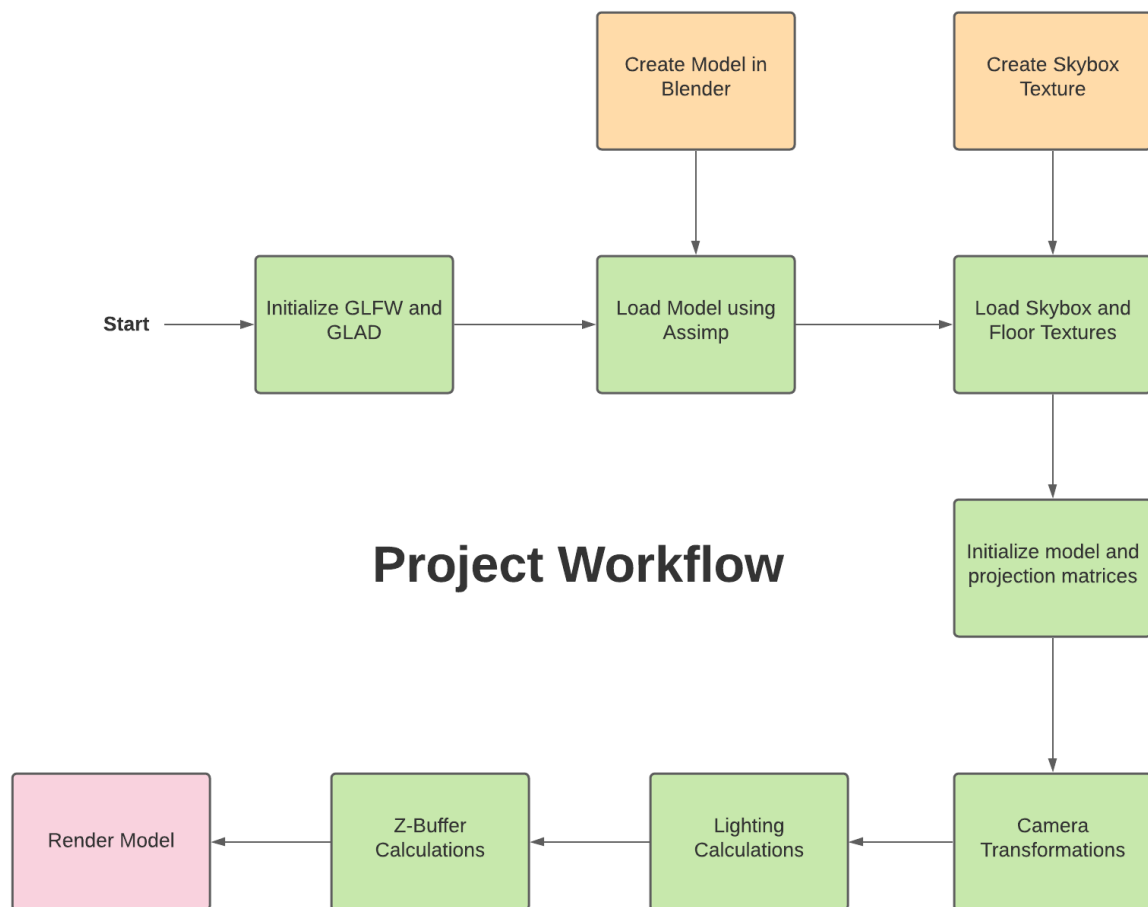
We consulted works from our previous seniors who had also used a similar workflow and graphics pipeline for rendering their model. Links to some previous works (available online) which we consulted for reference are listed below,

- ICTC Building Modeling 073 Batch (Anusandhan Pokhrel, Ashim Sharma, Ashish Agarwal) - <https://github.com/ashimsharma10/Graphics>

- ICTC Building Modeling 074 Batch (Rabin Adhikari, Safal Thapaliya, Samip Poudel) - <https://github.com/rabinadkl/ICTC-modeling>
- Solar System Modeling 073 Batch - <https://github.com/sobitneupane/Graphics-Project>

Methodologies

Graphical Representation of Project Workflow



Algorithms and Implementations

1. Projection (Perspective)

We use the following 4x4 matrix to transform our vertices or 3D points as seen from the perspective of the center of projection.

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

Where,

fov = field of view angle (45 degree for our project)


far = distance of far plane of the projection frustum from origin

near = distance of near plane of the projection frustum from origin

2. Matrix Transformations

We use the **translation** and **scaling matrices** to transform the model world space coordinates.

Translation matrix


$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + dx \\ y + dy \\ z + dz \\ 1 \end{bmatrix}$$

$$\begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x * sx \\ y * sy \\ z * sz \\ w \end{bmatrix}$$

3. Phong Lighting

Phong lighting model uses phong shading to approximate the local illumination at the surface of the object. It consists of Ambient color, Diffuse reflection and specular reflection. Diffuse reflection component is the directional impact of a light on the surface.

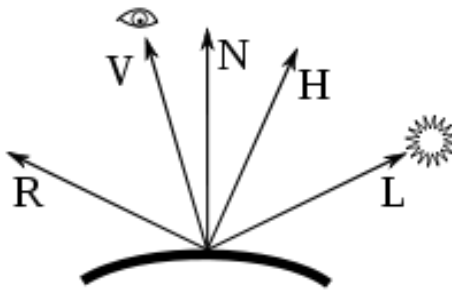


Fig. Phong reflection

Illumination at point p,

```
total_illum = Ia * Ka ;  
// Ia    -> Ambient light strength  
// Ka -> ambient reflection constant  
for (each light source in scene)  
{  
    // Calculate the diffuse component at point p  
    diff_comp = Kd * dot(L*n) * Id ;  
// Id    -> diffuse intensity of light source  
// Kd -> the diffuse constant of the object  
    spec_comp = Ks * pow( dot(R*V),shine) * Is;  
// Is    -> specular intensity of the light source  
// Ks -> specular constant of the material  
// R     -> the reflected vector of the light along the  
normal
```

```
// V      -> the view vector from the camera/eye to the
//          fragment position
// shine -> shininess factor of the object
    total_illum += diff_comp + spec_comp;
}
```

Ambient component is independent of the light source and represents the environmental property.

4. Z-buffering

The Z-buffer algorithm proceeds as follows :

Initially the depth of all pixels are considered to be very high.

```
depth(i, j) = infinite // Max length .. In context of opengl
it is set to 1.0f, since OpenGL uses Normalized Device
Coordinates

// Initialize the color value for each pixel to the
background color
pixel(i, j) = background color

// For each polygon, do the following steps :
for (each pixel in polygon's projection)
{
    z = depth of the current pixel
    if (z < depth(i, j)) // d(i,j) is the depth of current
pixel stored in depth buffer
    {
        depth(i, j) = z;
        pixel(i, j) = color;
    }
}
```

```
}
```

In our context, since OpenGL doesn't allow us to read and write directly from a depth buffer, we did a workaround. We first rendered the depth values on a texture and sampled the texture to obtain the depth value of the current pixel.

Z buffering is the embedded stage of OpenGL in the modern pipeline. Our approach works but due to floating point precision errors, the objects at far distance may flicker and this manual process is obviously slower than the one done by OpenGL itself.

5. Camera

Modern pipeline of OpenGL, in itself, has no concept of a camera. Basically, we try to simulate a camera in OpenGL by transforming the world in reverse of the applied transformation. That is, if we want to simulate a camera moving behind, we translate the whole world in the forward direction.

In 3D, we are concerned with orientation of the camera and translation of the simulated camera from the origin. So, we first translate the camera back to origin and re-orient it to match the world coordinate axes. For OpenGL, we assume that the camera is facing toward the negative Z-axis and is located at the origin.

If the camera location is given by the translation matrix **T** and the orientation of camera is given by **R**, the the whole world is transformed by the matrix

$$\mathbf{M_c} = \text{inv. } \mathbf{R} * \text{inv. } \mathbf{T}$$

```

Mat4x4 LookAtMatrix(vec3 cameraPos, vec3 target, vec3 up)
{
    for_vec = (cameraPos - target).unitVec();
    right = cross(up, for_vec).unitVec();
    w_up = cross(forward_vec, right).unitVec();
    Mat4 id{1.0f};
    id[0][0] = right.x;
    id[1][0] = right.y;
    id[2][0] = right.z;
    id[3][0] = -cameraPos.dot(right);

    id[0][1] = w_up.x;
    id[1][1] = w_up.y;
    id[2][1] = w_up.z;

    id[3][1] = -cameraPos.dot(w_up);

    id[0][2] = for_vec.x;
    id[1][2] = for_vec.y;
    id[2][2] = for_vec.z;
    id[3][2] = -cameraPos.dot(for_vec);
    return id;
}

```

Languages and Tools

Programming Language - C/C++,

Build Tool - CMake

Version Control System - Git

Editor - Emacs

The majority of our code is written in **C++**, with a few **C-style header files** imported for OpenGL usage (more on that later). We have also used the build tool **CMake** to make the entire project cross-platform (Linux, Windows, Mac). For version control, we have used **Git** and our project is available on **GitHub**.

OpenGL

OpenGL (Open Graphics Library) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a GPU, to achieve hardware-accelerated rendering. We have used the **Glad** GL Loader-Generator library for better functionality and extension of the OpenGL specification. OpenGL allowed us to do the bulk of the rendering work through vertex and fragment shaders - including lighting and surface rendering, SIMD-based matrix transformations, perspective projection and more.

GLFW

GLFW is an Open Source, multi-platform library for OpenGL development on desktop. The GLFW API has been used in our project for creating windows, contexts and receiving keyboard and mouse input/events.

Blender + Assimp (3D Modeling)

The rocket model was first rendered using **Blender 3D** which is a free and open source 3D creation suite supporting the entirety of the 3D pipeline. All materials and textures for the model are applied and viewed within Blender and the vertex data are exported as object files. We loaded our model into our code using the Open Asset Import Library, or **Assimp** in short.

Output and Result

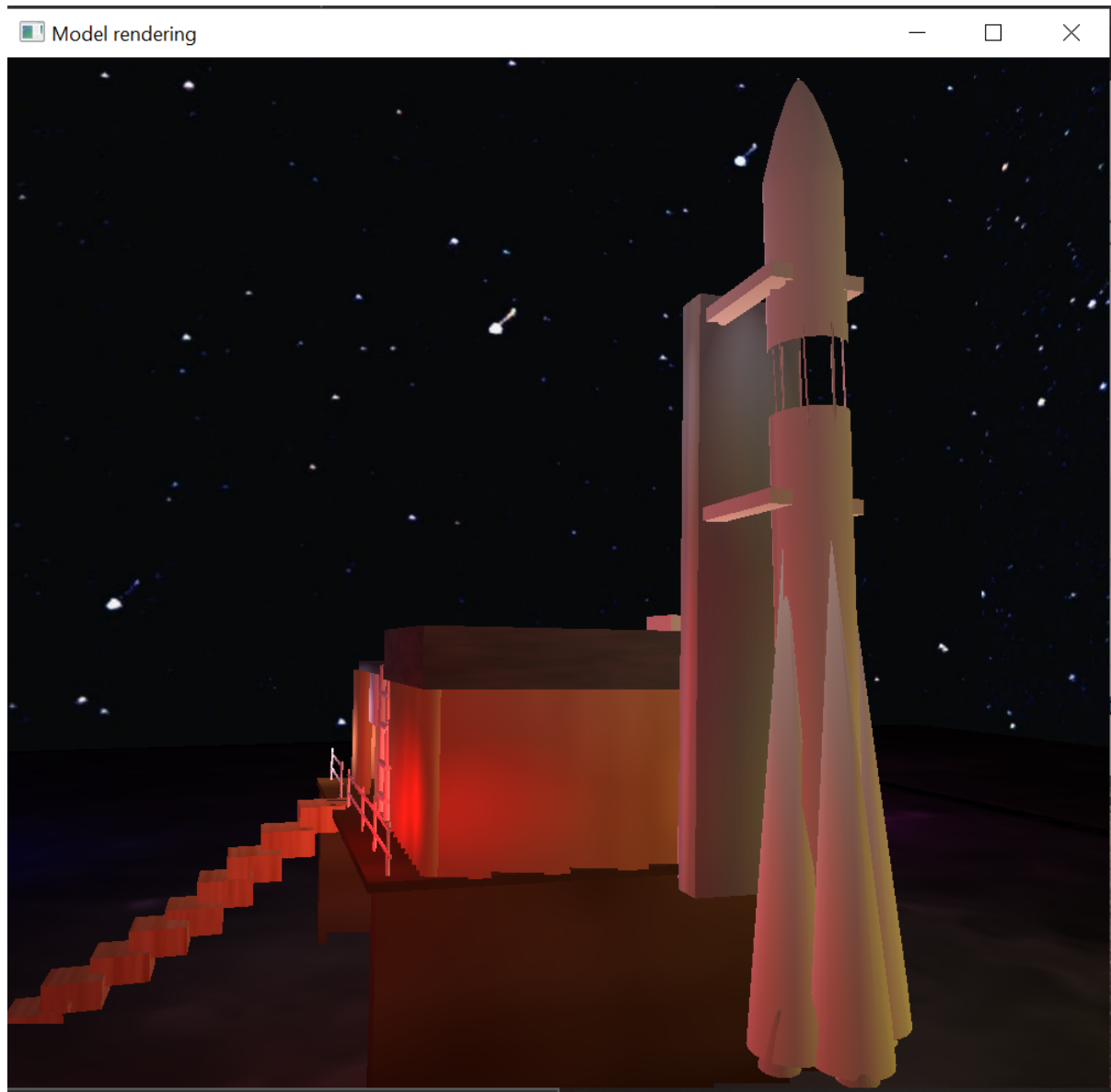


Fig. Side view of the modal

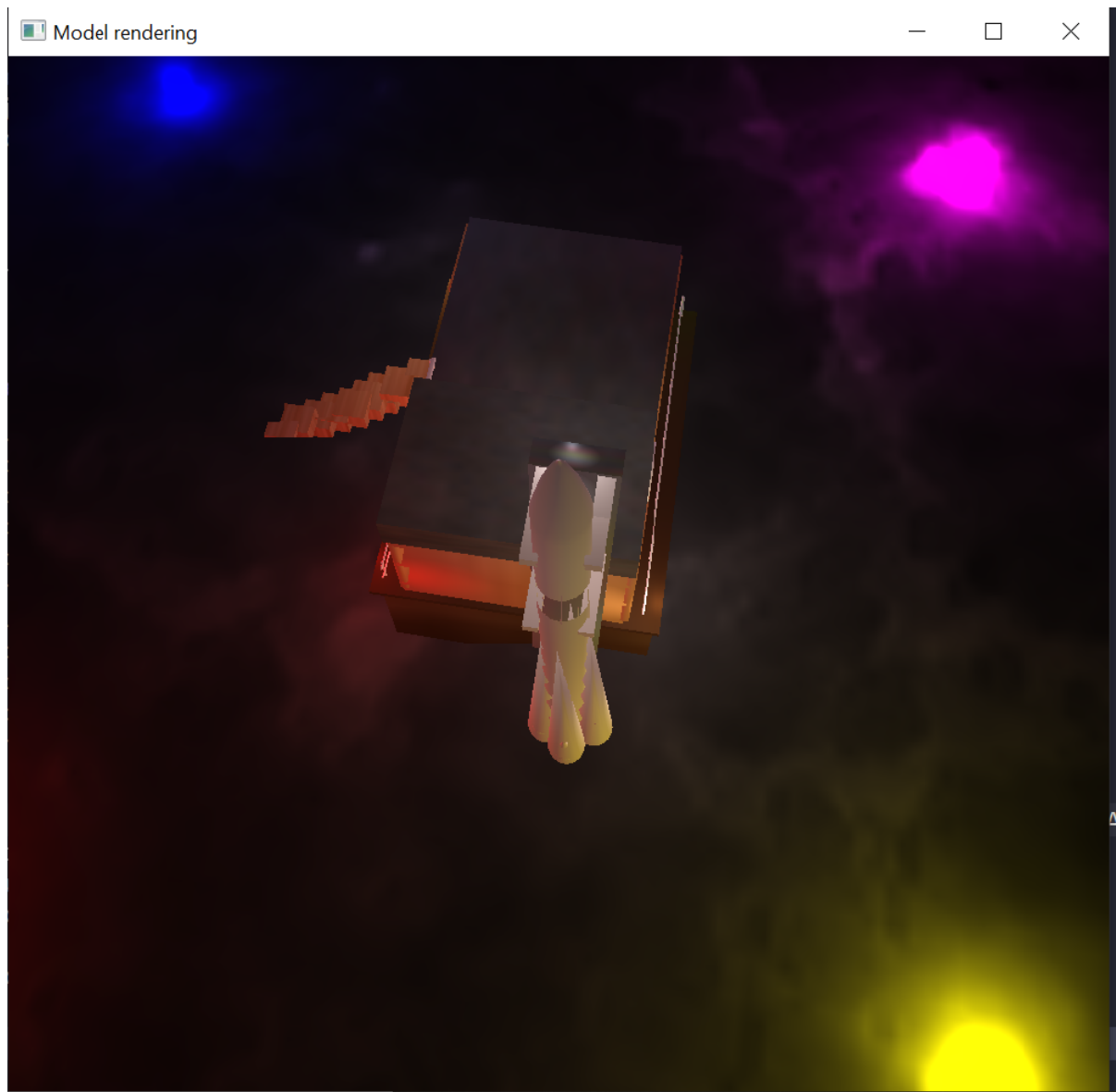


Fig. Top view of the Rocket with corner lights

Conclusion

We used the concepts of computer graphics to model and render Vostok-1 Rocket on the computer. We used Blender for modeling, we used Assimp for import and we used C++ language for rendering with the help of various libraries under the OpenGL specification. In this course, we used the concepts and algorithms that we learned in computer graphics class. We have registered the transformations, managed the lighting and added reflections in the water. In the process of working on this project, we learned computer graphics and learned to work as a team.

Limitations and Future Works

In this Graphics project, we modelled rockets and learned a lot about Computer Graphics. We put a considerable amount of effort into doing this project. Yet it has its limitations and possible further improvements that could be applied.

Limitations are :

- Loading and rendering models with extremely high amount of vertices slows down it considerably
- Depends upon dependencies like GLFW, glad and Assimp
- The z buffering, we implemented manually, works but is very slow compared to the one done by OpenGL itself. It is expected since z buffering is the embedded stage in OpenGL and OpenGL does it's things best.
- Model loaded is static i.e there's no any kind of animation
- Z fighting due to floating point precision error at greater depth

Similarly, limitations can be overcome and it could be further improved by :

- Program to load a specific format of a model can be written manually and might increase the performance since Assimp employs generic interfaces to load a wide variety of models.
- By rigging and creating bone armature in Blender, the launch process of the rocket can be animated. It might take a lot of effort though.
- Dependencies could be reduced by using platform specific APIs and native OS functionality.

References

Learn OpenGL - <https://learnopengl.com/>

<https://www.scratchapixel.com/>

<https://www.khronos.org/registry/OpenGL-Refpages/gl4/>

<https://thebookofshaders.com/>

https://en.wikipedia.org/wiki/Phong_reflection_model