Vostok Rocket

Graphics Project, 5th Semester, 075BCT

Pranjal Pokharel (075BCT061)

Tilak Chad (075BCT094)

Udeshya Dhungana (075BCT095)

Table of contents

01

Introduction

Overview and objectives

+ Overview

Project Title : Vostok Rocket Modeling

Working Period : 29th June, 2021 to 27th August, 2021

Motivations: Implement algorithms and rendering techniques we learned as a part of our curriculum including,

- Polygon rendering,
- 2D and 3D transformations,
- Surface rendering,
- Visible surface detection (z-buffer),
- Illumination and more.

+ Objectives

1. To learn about **OpenGL specification**.
2. To learn about the **Graphics Pipeline** - shaders, viewports, batch rendering, matrix and view transformations and more.
3. To understand concepts behind lighting such as ambient, diffuse and specular reflections as a part of the **Phong shading model**.
4. 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.
5. To learn **Blender 3D** for designing professional 3D models and animation.
6. To learn to work in a team using **Version Control System (VCS)**.

+ Work Distribution

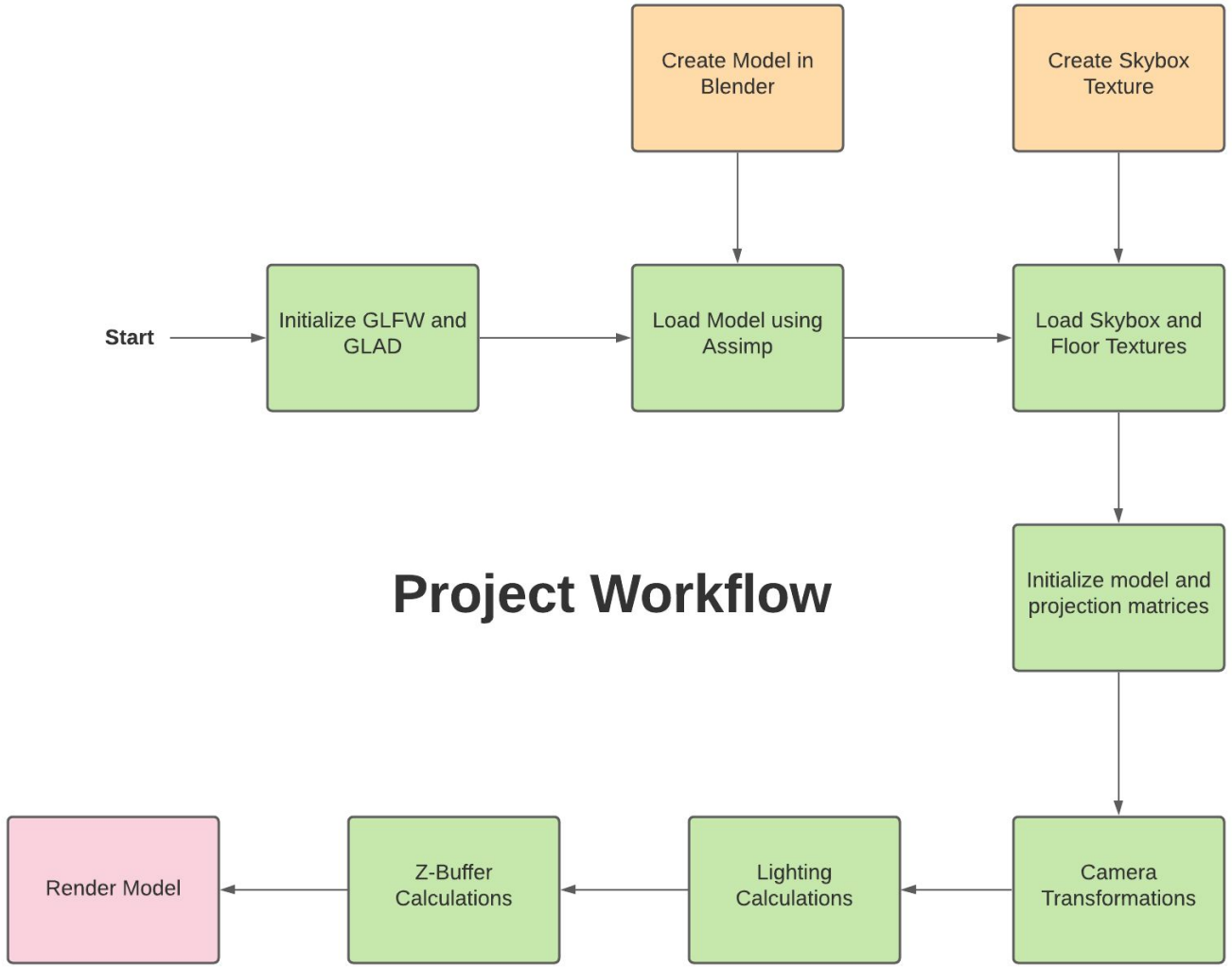
Team Member	Work
Tilak Chad (075BCT094)	Z-buffer, Camera, Maths Library, Model Loading
Pranjal Pokharel (075BCT061)	Model Loading, Lighting, Cubemap, Maths Library
Udeshya Dhungana (075BCT095)	Blender 3D, Lighting, Maths Library

02

Implementation

Basic theory, tools and project workflow

Workflow


Create Model in Blender
Create Skybox Texture
Start
Initialize GLFW and GLAD
Load Model using Assimp
Load Skybox and Floor Textures
Project Workflow
Initialize model and projection matrices
Render Model
Z-Buffer Calculations
Lighting Calculations
Camera Transformations

Tools

+ Tools

Language (C++/C)

Majority of code written in C++, some C header files

OpenGL (Glad)

API, hardware-accelerated rendering through GPU

Blender 3D

Free and open source 3D creation suite

GLFW

Library for OpenGL development - window, contexts, keyboard input

Assimp

Open Asset Import Library, importing model into code

CMake

Build automation, testing, packaging and installation

Algorithms and Implementations

Translation

- Move the object in the direction of basis vectors and their combinations.
- Translate the model within the **world space**.
- **tx** = change in object's x coordinate

  ty = change in object's y coordinate

  tz = change in object's z coordinate

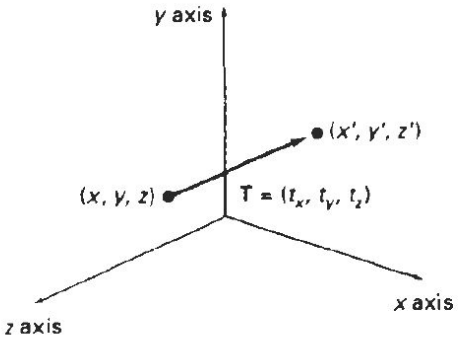
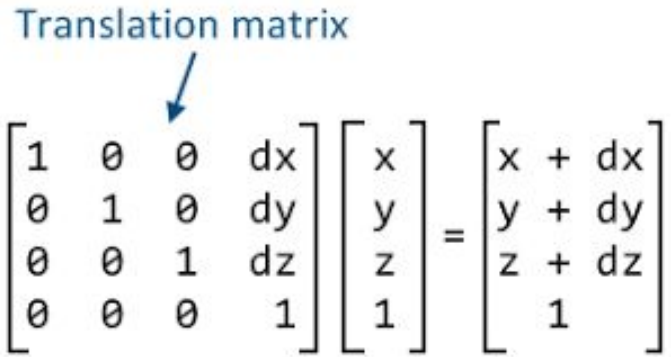


Figure 11-1
Translating a point with translation vector $\mathbf{T} = (t_x, t_y, t_z)$.

+ Translation

```
constexpr Mat4 translate(const Vec3<T> &vec) {
  Mat4 id{1.0f};
  id[3][0] = vec.x;
  id[3][1] = vec.y;
  id[3][2] = vec.z;
  return id * *this;
}
```

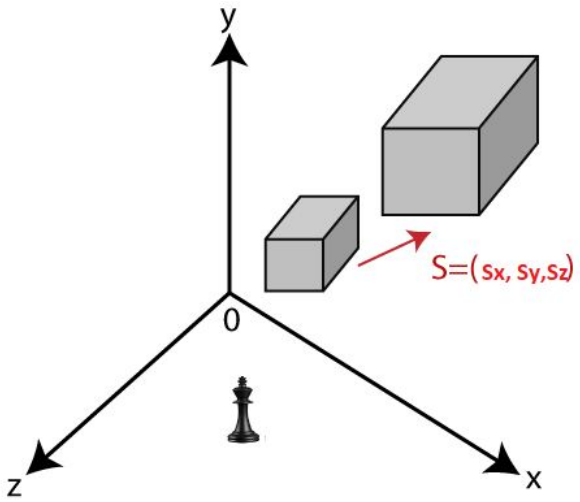


+ Scaling

- Scale the object with scale factors.
- Scale factors are arranged in the **principal diagonal.**
- **sx** = scale factor for x-coordinate

  sy = scale factor for y-coordinate

  sz = scale factor for z-coordinate



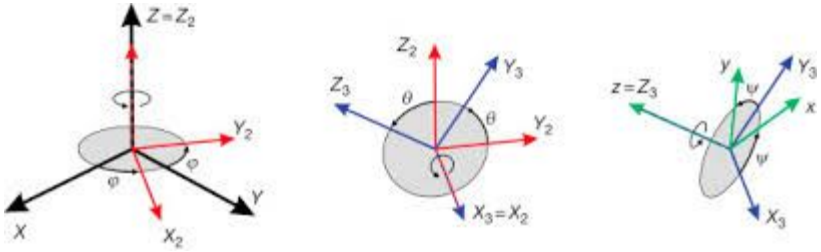
+ Scaling

```
constexpr Mat4 scale(const Vec3<T> &vec){
  Mat4 id{1.0f};
  id[0][0] = vec.x;
  id[1][1] = vec.y;
  id[2][2] = vec.z;
  id[3][3] = 1;
  return id * *this;
}
```

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

Rotation

- 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 have avoided it in our project.



+ Rotation

```
constexpr Mat4 rotateX(float angle) {
  Mat4 id{1.0f};
  id[1][1] = std::cos(angle);
  id[1][2] = std::sin(angle);
  id[2][1] = -id[1][2];
  id[2][2] = id[1][1];
  return id * *this;
}

constexpr Mat4 rotateY(float angle) {
  Mat4 id{1.0f};
  id[0][0] = std::cos(angle);
  id[0][2] = -std::sin(angle);
  id[2][0] = -id[0][2];
  id[2][2] = id[0][0];
  return id * *this;
}

constexpr Mat4 rotateZ(float angle) {
  Mat4 id{1.0f};
  id[0][0] = std::cos(angle);
  id[0][1] = std::sin(angle);
  id[1][0] = -id[0][1];
  id[1][1] = id[0][0];
  return id * *this;
}
```

X-axis rotation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y-axis rotation

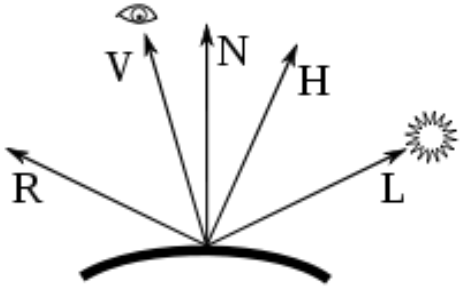
$$\begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z-axis rotation

$$\begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 & 0 \\ \sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Phong Shading

- Interpolation technique for surface shading.
- Calculate lighting information per pixel based on interpolated normals.
- **Ambient** - Surrounding light, constant for a given scene.
- **Diffuse** - Directional impact of light source on surface normal, dot product of directional light and the surface normal
- **Specular -** mirror-like reflection, depends on the dot product of the reflected vector of light along the normal and the view vector.

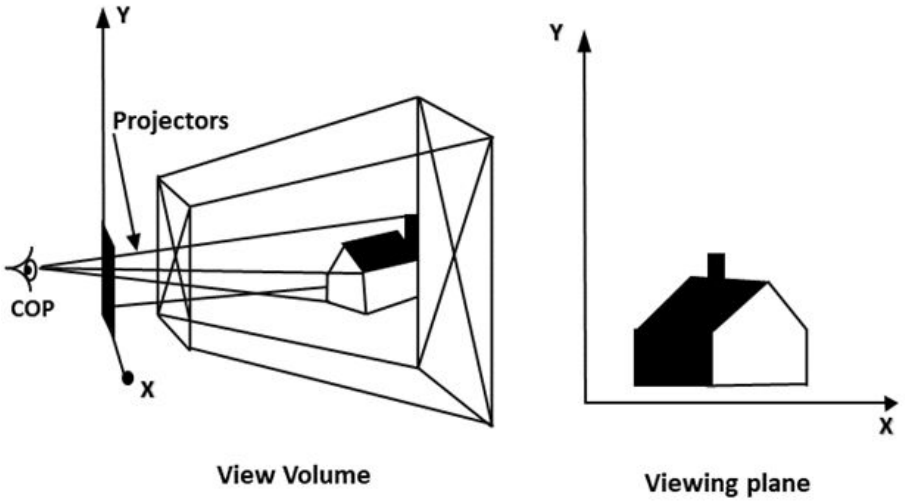


+ Phong Shading

```
for (int i = 0; i < NR_LIGHTS; i++){
  // ambient
  vec3 ambient = light[i].ambient * textureDiffuse;

  // diffuse
  vec3 lightDir = normalize(light[i].position - FragPos);
  float diff = max(dot(norm, lightDir), 0.0);
  vec3 diffuse = light[i].diffuse * diff * textureDiffuse;

  // specular
  vec3 reflectDir = normalize(lightDir + viewDir);
  float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);
  vec3 specular = light[i].specular * (spec * textureSpecular);

  // attenuation
  float dist = length(light[i].position - FragPos);
  float attenuation = 1.0 / (light[i].constantAtten + light[i].linearAtten * dist + light[i].quadraticAtten * dist * dist);
  ambient *= attenuation;
  diffuse *= attenuation;
  specular *= attenuation;

  // output
  result += (ambient + diffuse + specular);
}
FragColor = vec4(result, 1.0);
}
```

+ Projection

- **Projection** - Perspective Projection
- **Center of Projection**, distance from COP to the projection plane is finite unlike in parallel projection
- Distant objects appear smaller than nearer objects.
- Anything outside the perspective frustum is clipped, only objects within near and far plane are visible.



+ Projection

```
Mat4 perspective(float aspect_ratio, float fovy = 45.0f,
                 float near_plane = 0.1f, float far_plane = 100.0f) {
  Mat4 id{1.0f}; // Need c++17 conformance compiler
  float tan_fovy = std::sin(FMath::DegreeToRadians(fovy)) /
                   std::cos(FMath::DegreeToRadians(fovy));
  float top = near_plane * tan_fovy;
  float right = aspect_ratio * top;

  id[0][0] = near_plane / right;
  id[1][1] = near_plane / top;
  id[2][3] = -1;
  id[3][2] = -2 * far_plane * near_plane / (far_plane - near_plane);
  id[2][2] = -(far_plane + near_plane) / (far_plane - near_plane);
  id[3][3] = 0;
  return id * *this;
}
```

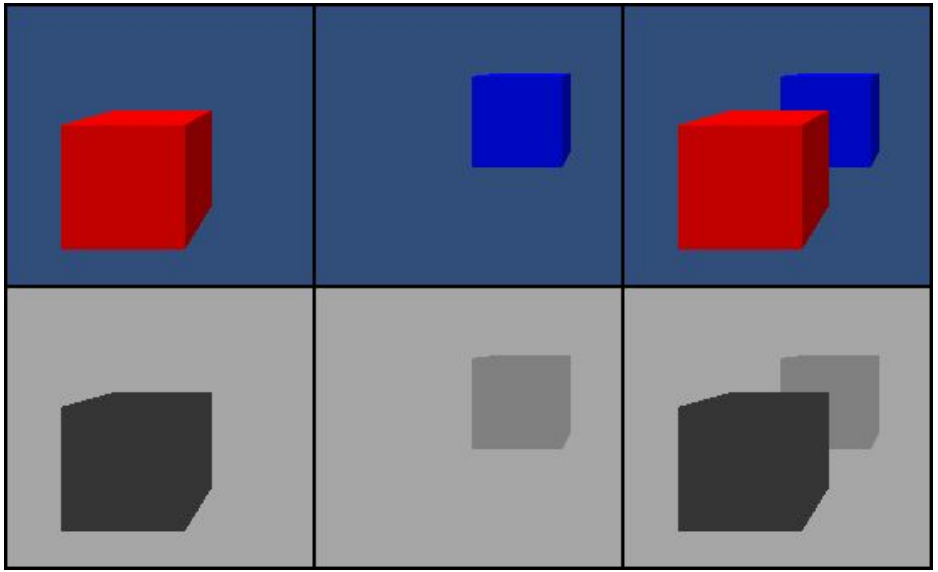
$$\begin{bmatrix} \frac{1}{aspect * \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2 * far * near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

+ Z Buffer

- **Z-Buffer or depth buffer** is used to represent depth information of objects in 3D space from a particular perspective.
- 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.

+ Z Buffer

```
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;
    }
}
```



03

Output

Demo and Discussion

Demo

04

Conclusion

Remarks and future improvements

+ Current Limitations

1. Loading and rendering models with **extremely high amount of vertices slows** down it considerably.
2. Depends upon **dependencies** like GLFW, glad and Assimp.
3. We implemented the **z-buffer manually**. Although it works efficiently, it is **slow** compared to the **in-built functionality offered by OpenGL**. It is expected since z buffering is an embedded stage in OpenGL and OpenGL does it's things best.
4. Model loaded is static i.e there's **no animation**.
5. **Z fighting** due to floating point precision error at greater depth

+ Future Improvements

1. 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.
2. 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.
3. Dependencies could be reduced by using **platform specific APIs and native OS functionality**.

Thank You