TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

A
REPORT ON

AIRPLANE MODELLING

Submitted By:
Milan Shrestha (075BCT050)
Prabin Paudel (075BCT060)
Rahul Shah (075BCT063)

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
LALITPUR, NEPAL

TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

A
REPORT ON
AIRPLANE MODELLING

Submitted To:
Asst. Prof Anil Verma

By:
Milan Shrestha (075BCT050)
Prabin Paudel (075BCT060)
Rahul Shah (075BCT063)

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
LALITPUR, NEPAL

Abstract

We used different graphical algorithms to render a 3D model of A380 in this project. The algorithms are implemented using user-defined functions and OpenGL. First of all, we created a glfw window. To model A380 we realized points using Blender. The normals to all the vertices were also found out by this software. The coordinates along with the normal were manually used to define the outlines of the plane and the lighting mechanism. To realize the 3D object into 2D plane, we defined three different matrices. We create a transformation matrix for each of these steps: model, view and projection matrix and multiply them in order to get a clipped matrix. The view matrix transforms all the world coordinates into view coordinates that are relative to the camera's position and direction. The transformation matrix is integrated with the rendering pipeline to get the final output. We defined a camera by stating the position in world space, the direction it's looking at, a vector pointing to the right and a vector pointing upwards from the camera. Camera is implemented to give three different views; first static, second following the plane and third is the cockpit view. We used an algorithm for one point perspective projection. In the same way, other important geometrical transformations such as translations, rotations and reflections are implemented. In order to give a more realistic view, we use three different light sources; point, spot and direction in the scene and by comparing its relative position against the position of the light source, we calculated its intensity. And visible surfaces were detected using a depth buffer algorithm implemented internally by OpenGL. Also a control mechanism is introduced to move the small plane using a key pressed event.

Contents

List of Figures

1 Introduction

Computer Graphics are the graphics created by using computers and deals with the representation of image data by a computer specifically from specialized graphic hardware and software. Computer graphics are extensively used for various computer games, 3D rendering/Modeling and virtual reality. Accordingly, the project “Airbus A380 Modeling” provides a rendered 3D model with all sorts of transformation and lighting effects. The project is developed in Visual Studio code, using OpenGL, a hardware accelerated API.

The main focus of the project is the creation and transformation of the Airplane model. The model is rotated, translated and scaled using 3D transformations, a camera with 3 different camera angles (Static, Following and Cockpit View) and projection plane and rendered with essential texture and color of calculated light intensity in each pixel lying inside the model. We considered triangles as the basic unit of 3D model and pixel as the basic unit of each triangle. Similar approach can be attempted to model other real world objects. Thus, 3D modeling is done in Computer Graphics.

1.1 Background

1.1.1 OpenGL

Open Graphics Library is a cross-language, cross-platform application programming interface (API) which is used to interact with a GPU, to achieve hardware-accelerated rendering. It generates 2D as well as 3D vector graphics. It consists of numerous functions and command that can be utilized to specify the objects and operations needed to produce interactive three-dimensional applications. It is cross-platform and is most commonly used in professional graphics applications.

OpenGL is considered to be the only universal standard graphics library. It is capable of creating lines, points, triangles, and other complex geometric figures. OpenGL provides a powerful but primitive set of rendering commands, and all higher-level drawing such as three-dimensional modelling must be done in terms of these commands. The desired model should be built through a small set of geometric shapes such as points, lines and polygons. However, a number of libraries are built on top of OpenGL which simplify the programming tasks.

1.1.2 Window (pyGLFW)

GLFW is one of the libraries specifically designed for use with OpenGL. It only comes with the absolute necessities: window and context creation and input management. It offers the most control over the OpenGL context creation among the OpenGL libraries. GLFW is originally written in C, specifically targeted at OpenGL providing the bare necessities required for rendering goodies to the screen. Along with providing us the window GLFW also provides the ability to take in external commands from the user in real time which affects the displaying property. This feature is extensively used in our project to get a 3D object and rotate and translate it using the inputs from keyboard and mouse.

pyGLFW is a set of python bindings to the successful cross-platform GLFW. It is intended as a powerful and lightweight alternative to pygame for python OpenGL application development.

1.2 Objectives

The main objectives of this project are:

1. To understand different graphical algorithms
2. To generate virtual reality systems by including rendering, lighting and shading effects
3. To develop programming knowledge of Python
4. To learn about OpenGL standards
5. To explore the practical application of theoretical knowledge

1.3 Scope

The aim of this graphics project is to get familiar with a different computer graphics system, standards and algorithms. The scope of our project is vast in various real-world modeling, computer games, virtual reality, 3D printing and many more. With further research, our project can be used in computer simulation. This may prove effective for pilots who are required to practice emergency procedures semi-annually.

Thus, “Airbus A380 Modeling” seems quite a useful project with modern need and scope.

2 Literature Review

Graphics has been an important aspect in this era of user-friendly system for decades. Computer Graphics plays a vital role in relaying the information easily and unambiguously. Although graphics is developing alongside the technology, its pace lags far behind that of technology.

In this modern era of technology, graphics has its major role in as animation, game, virtual reality. It is not only limited to entertainment sectors but has its importance on other sectors like 3D printing, formula visualization and modelling of engines, molecules, atoms as well as on medicinal sector for X-rays, Ultrasound, MRI where a 3D model is processed and displayed in 2D plane.

There are many large companies who devote on improving the graphics. However, their solutions are expensive and have high requirements. They are memory and time intensive. Many devices cannot fulfill their specification and there is no suitable low specification alternative for 3D model viewer. This project focuses on displaying a 3D model of an airplane model without using time and memory intensive software but instead Python and its libraries. It doesn't require high hardware or software specifications. Any computer with Python preinstalled can run this program to display a 3D model.

2.1 Related Theories

2.1.1 OpenGL Objects

OpenGL Objects are responsible for transmitting data to and from the GPU. OpenGL objects must first be created and bound before they can be used. There are several types of OpenGL Objects. For example, a **Vertex Buffer Object** can store vertices of a character. Whereas a **Texture**, another kind of OpenGL Object, can store image data.

Objects in OpenGL can be separated into two different categories:

1. Regular Objects

These types of objects contain data. A list of regular objects are:

- Buffer Objects
- Texture Objects

2. Container Objects

These types of objects do not contain any data. Instead, they are containers for regular OpenGL objects. A list of container objects is:

- Framebuffer Objects
- Vertex Array Objects

2.1.2 Buffer Objects

A Buffer object inherits the states and functionalities from an OpenGL object. A buffer object is an OpenGL object that contains unformatted data. Buffer objects are stored in the GPU (Graphics Processing Unit), which provides fast and efficient access.

1. Vertex Buffer Objects

A Vertex Buffer Object (VBO) is a Buffer Object. When a buffer object's binding point is set to GL_ARRAY_BUFFER, the buffer object behaves as a Vertex Buffer Object.

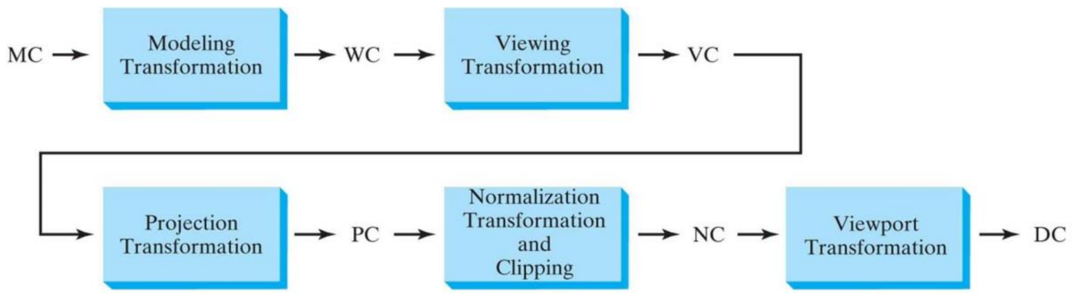
A VBO is a buffer object that represents storage for vertex data. This data can be a character's vertex, normal, texture coordinate, etc. OpenGL can't read vertex data in CPU memory. Vertex data must be sent to the GPU before it can be used in the rendering pipeline. A VBO is able to take the data stored in the CPU and transmit it to the GPU.

Even though a VBO lives in the GPU, there is no connection between a VBO and a Shader program. This connection is established through the function glVertexAttribPointer(). The glVertexAttribPointer() informs the shaders where to get the data found in the VBO and how that data should be interpreted.

2. Vertex Array Objects

A Vertex Array Object (VAO) is an OpenGL object. A VAO is a container for Vertex Buffer objects (VBO). A VAO is a container that stores all the states needed for rendering. It stores the information of vertex-attribute as well as the buffer object.

2.1.3 Coordinate Systems



1. Local Space (Model Coordinate)

Local Space is the coordinate space with local coordinate system of objects and represent the initial position and orientation of objects before any transform is applied. Local coordinate system simply means the coordinate system where the model was made in.

2. World Space (World Coordinate)

This is the space with the coordinate system of the virtual environment where the model is going to be in. This is the coordinate space where we want our objects transformed to in such a way that they're all scattered around the place (preferably in a realistic fashion). The coordinates of our object are transformed from local to world space with the model matrix.

For every Model in the scene, it is going to have a model matrix and this matrix is going to transform the model from object space to world space.

3. View Space / Eye Space (View Coordinates)

The view space is usually referred to as Camera. In this coordinate system everything is relative to camera's position and the camera never moves instead the world moves around the camera.

To get the model into the view/camera coordinate system, we are going to have a view matrix given by LookAt() function and then uploaded to shader program.

4. Projected Space (Projected Coordinate)

A computer monitor is a 2D surface. In this space we apply projection matrix to view coordinate whose responsibility is to take 3D data and project it into 2D space. Basically, we have 2 types of projection:

orthographic projection: depth does not matter, parallel remains parallel
perspective projection: used to give depth to the scene (a vanishing point)

To get the eye space coordinate into projected space coordinate, we are going to have a create_prespective_projection() function.

5. Clip Space (Normalized Coordinate)

We need to transforms all vertex data from the eye coordinates to the clip coordinates which is done with create_perspective_projection matrix.

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = M_{projection} \cdot \begin{pmatrix} x_{eye} \\ y_{eye} \\ z_{eye} \\ w_{eye} \end{pmatrix}$$

Then, these clip coordinates are also transformed to the normalized device coordinates (NDC) by dividing with *w* component of the clip coordinates.

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} x_{clip}/w_{clip} \\ y_{clip}/w_{clip} \\ z_{clip}/w_{clip} \end{pmatrix}$$

Its end result is it gives Normalized Device Coordinates. (NDC's between -1 to 1) and it is assigned to gl_Position.

6. Viewport Transformed Space (Device Coordinate)

The result from scaling and translating Normalized Device Coordinates is controlled by the viewport transformation. They are controlled by the parameters of the viewport we defined:

glViewport(): to define the rectangle of the rendering area where the final image is mapped.

In our Project it is mainly called when we perform window resize.

NOTE: Basically, we are multiplying each vertex of the model by a matrix which is going to give a new point. This is really a tedious task.

The good thing is our graphics card, which is really efficient with matrices. So, matrix is used mostly even for basic transformation.

2.1.4 Phong Illumination Model

Lighting in the real world is extremely complicated and depends on many factors which cannot be calculated accurately using 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. One of those models is called the Phong Lighting model. The major building blocks of the Phong lighting model consist of 3 components: ambient, diffuse and specular lighting. The intensity of a point on a surface is taken to be the linear combination of these three components.

1. Ambient

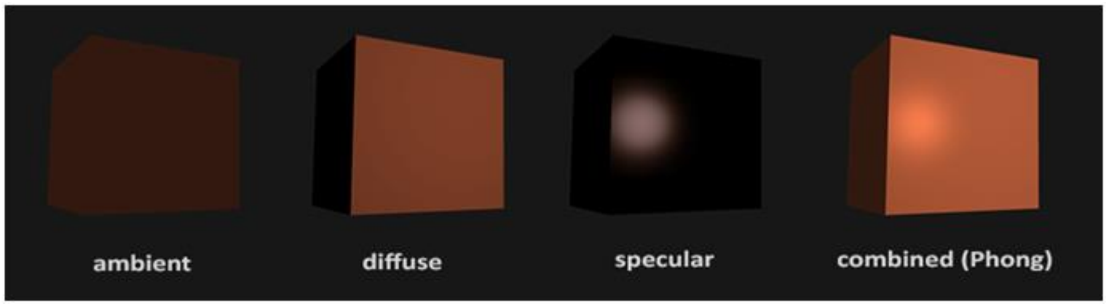
It is the minimum amount of light used to simulate global illumination. It occurs due to scattering of light from many light sources. One of the properties of light is that it can scatter and bounce in many directions, reaching spots that aren't directly visible; light can thus reflect on other surfaces and have an indirect impact on the lighting of an object. A very simplistic model of global illumination is ambient lighting.

2. Diffuse

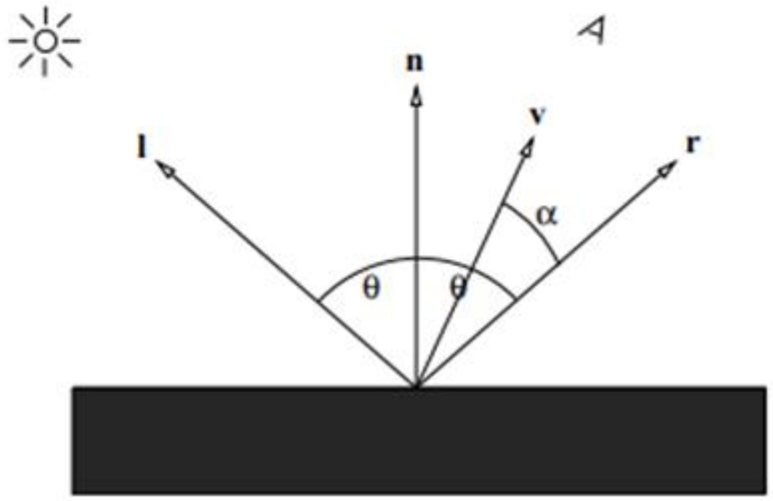
The way light "falls off" of an object. It simulates the directional impact a light object has on an object. This is the most visually significant component of the lighting model. The more a part of an object faces the light source, the brighter it becomes.

3. Specular

It simulates the bright spot of a light that appears on shiny objects. Specular highlights are more inclined to the color of the light than the color of the object.



Relevant vectors for Phong Shading



The shading of a point on a surface is a function of the relationship between the viewer, light sources, and surface. The following vectors are relevant to direct illumination. All vectors are assumed to be normalized to unit length.

Normal vector: A vector n n that is perpendicular to the surface and directed outwards from the surface.

View vector: A vector v that points in the direction of the viewer.

Light vector: A vector l that points towards the light source.

Reflection vector: A vector r that indicates the direction of pure reflection of the light vector.

The final color of an object is comprised of many things:

The base object color called a “material”

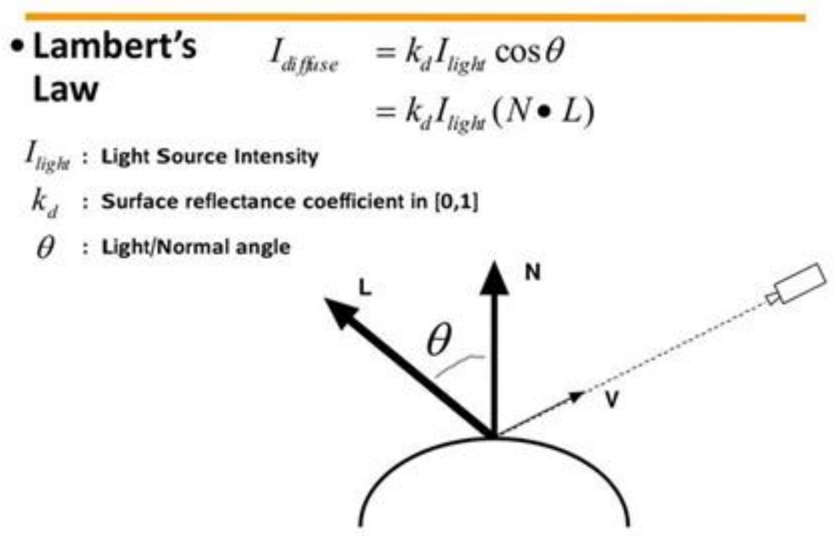
The light color

Any textures we apply

Components of Phong illumination model

1. Lambertian Reflectance (Diffuse component)

Light falling on an object is the same regardless of the observer's viewpoint. This follows Lambert's cosine law which states that the radiant energy from a small surface area dA is proportional to the cosine of angle Θ between surface normal and incident light direction



• Lambert's Law

$$I_{diffuse} = k_d I_{light} \cos\theta = k_d I_{light} (N \bullet L)$$

I_{light} : Light Source Intensity

k_d : Surface reflectance coefficient in [0,1]

θ : Light/Normal angle

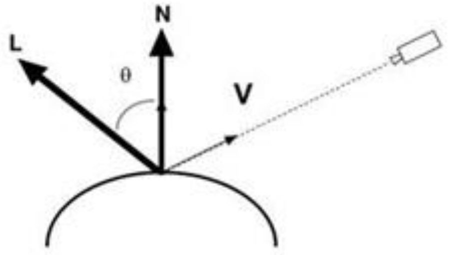
2. Ambient Light

It is independent of viewpoint. Object not exposed directly to a light source is visible with ambient light. Amount of ambient light incident on each object is a constant for all surfaces and over all directions. Ambient light produces flat shading which is not desirable in general, so scenes are illuminated with other light sources together with ambient light.

$$I_{d+a} = k_a I_a + k_d I_{light}(N \bullet L)$$

I_a : **Ambient light intensity (global)**

k_a : **Ambient reflectance (local)**



This is diffuse illumination plus a simple ambient light term

– **a trick to account for a background light level caused by multiple reflections from all objects in the scene**

3. Specular reflection

It describes the specular highlight and is dependent on viewpoint v. It is the result of total or near total reflection of incident light in a concentrated region around specular reflection angle Ө

One function that approximates specular falloff is called the *Phong Illumination* model

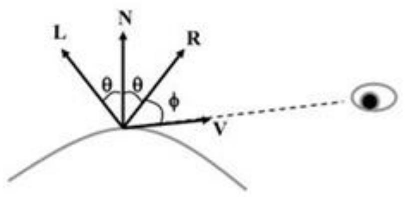
$$I_{specular} = k_s I_{light}(\cos\phi)^{n_{shiny}}$$

ϕ : **Angle between reflected light ray R and viewer V**

k_s : **Specular reflectance**

n_{shiny} : **Rate of specular falloff**

– **no physical basis, yet widespread use in computer graphics**

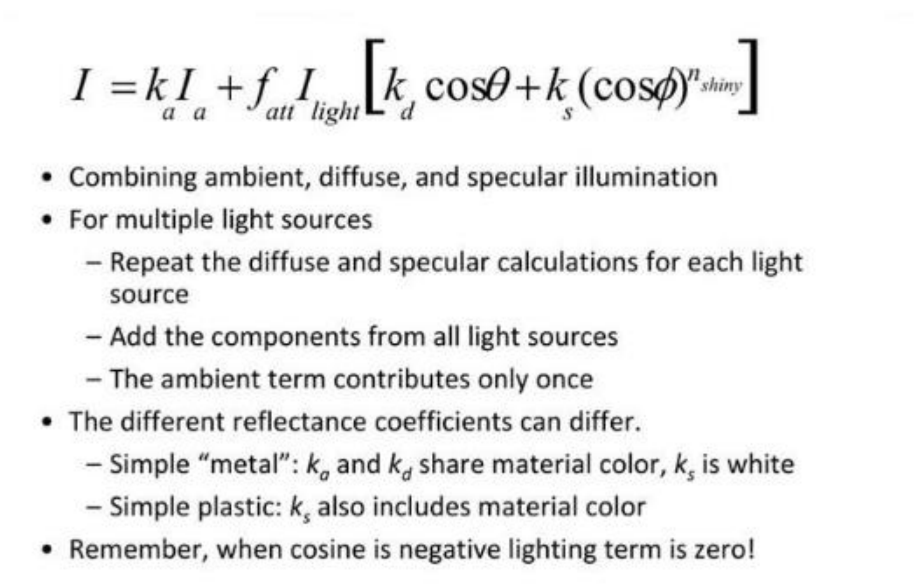


ns: specular reflection parameter (depends on surface)

Ø: ranges from 0 to 90^o (i.e. cosØ varies from 0 to 1)

Ideal reflector exhibits specular reflection in the direction of R only (i.e. Ø = 0) but for non-ideal case specular reflection is seen over a finite range of viewing positions.

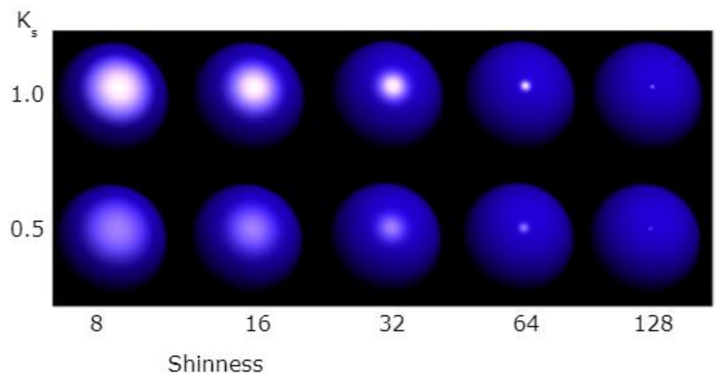
Putting it all together



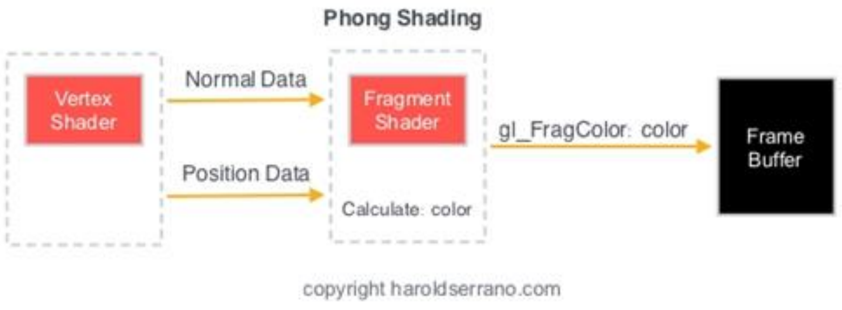
$$I = k_a I_a + f_{att} I_{light} \left[k_d \cos\theta + k_s (\cos\phi)^{n_{shiny}} \right]$$

- Combining ambient, diffuse, and specular illumination
- For multiple light sources
  - Repeat the diffuse and specular calculations for each light source
  - Add the components from all light sources
  - The ambient term contributes only once
- The different reflectance coefficients can differ.
  - Simple "metal": k_a and k_d share material color, k_s is white
  - Simple plastic: k_s also includes material color
- Remember, when cosine is negative lighting term is zero!

f_{att}: is the attenuation factor for real light – light intensity becomes weaker over distance. $f_{att} = 1/dL^2$ where dL is the distance from the surface point to the light.



Phong shading is a per-fragment color computation. The vertex shader provides the normal and position data as out variables to the fragment shader. The fragment shader then interpolates these variables and computes the color. Here is an illustration of the Phong Shading:



Lights sources used in our project

a) Point Light

These lights have a position in 3D space. They are sometimes called a Lamp. In this type light emanates from the light source in all directions. Distance d determines brightness (“attenuation”):

Intensity = 1/d2

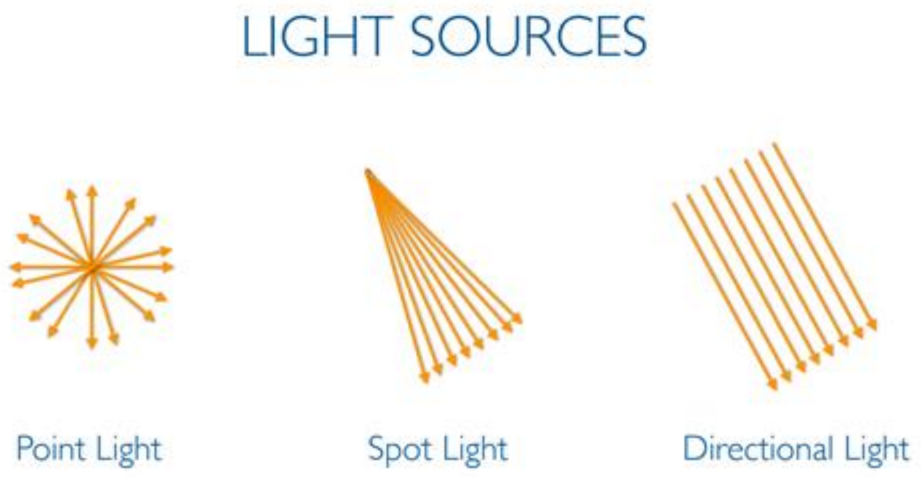
In our project it is represented by small spherical light sources of different colors.

b) Directional Light

The type of light source is infinitely far away. So, for this position of the light source doesn’t matter and direction is only used. All objects are lit evenly using directional lights. This type of light source is sometimes called a “Sun”. In our project it is represented by the background light which causes day and night effects.

c) Spot Light

In this type light starts at one point and spreads out as a cone with defined angle. It is described by position, direction and width of a beam. In our project it is represented by the headlight of the plane.



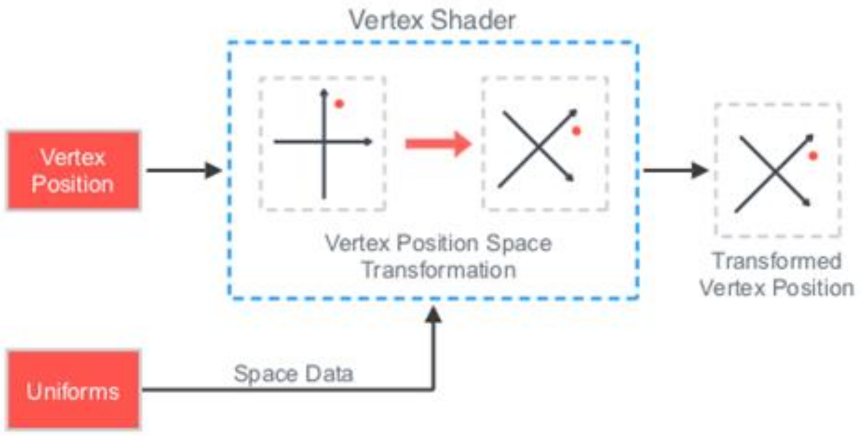
2.1.5 Shaders

A Shader is a program designed to run on some stage of a graphics processor. Shaders take the input from the previous pipeline stage (e.g., vertex positions, colors, and rasterized pixels) and customize the output to the next stage. 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:

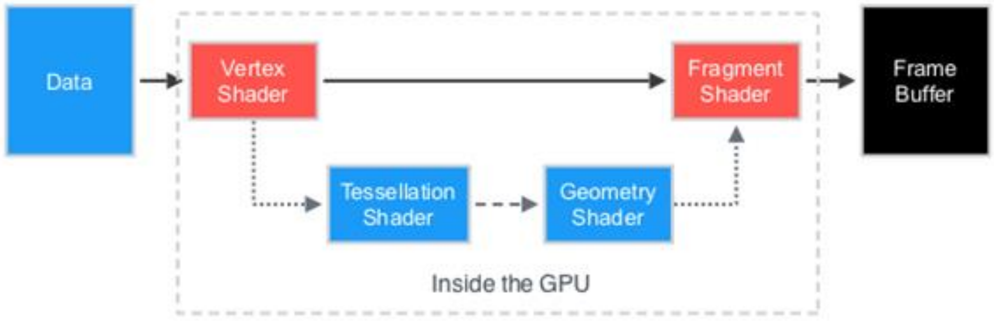
a) Vertex shader

The *Vertex Shader* is responsible for transforming the coordinate space of a model into Clip Space using 4D matrix multiplication. The vertex shader transforms each vertex's 3D position in virtual space (your 3d model) to the 2D coordinate at which it appears on the screen.



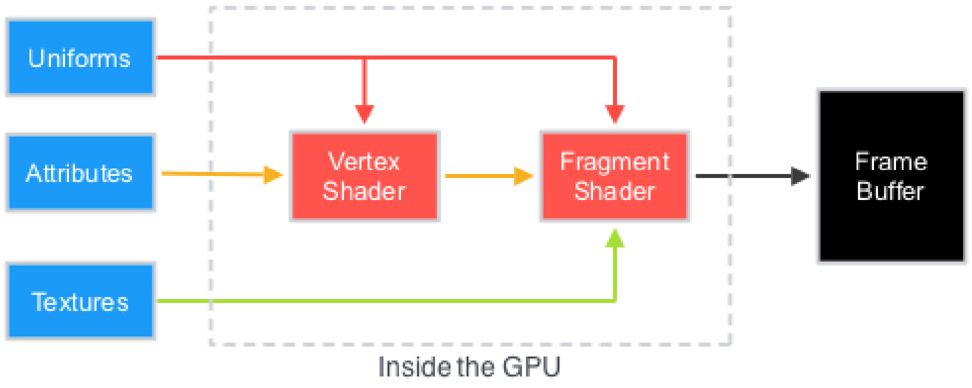
Vertex Shader
Vertex Position
Vertex Position Space Transformation
Transformed Vertex Position
Uniforms
Space Data

It is also responsible for forwarding these coordinates, to other shaders like Fragment shader.



Vertex Shader Inputs

Inputs to vertex shaders can be Attributes and Uniforms. Both attributes and Uniforms are treated as Read-Only variables by the vertex shader.

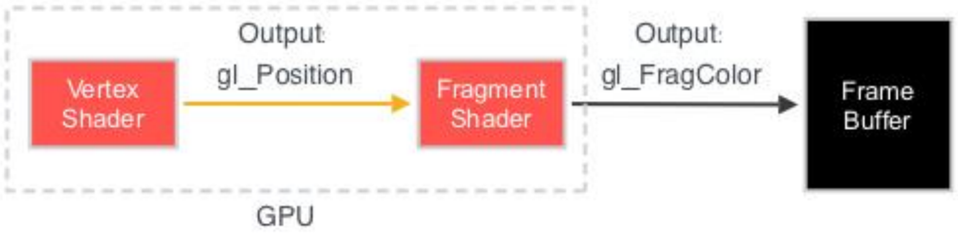


Attribute variables, such as *positions, normals and uv-coordinates* for each vertex, are considered read-only to the vertex shader.

Uniform variables are constant across a graphics primitive and are read-only to **all** shaders. Thus, Uniform variables can be read by the Vertex Shader, Tessellation Shader, Geometry Shader and Fragment Shader but can't be directly modified.

Vertex Shader Outputs

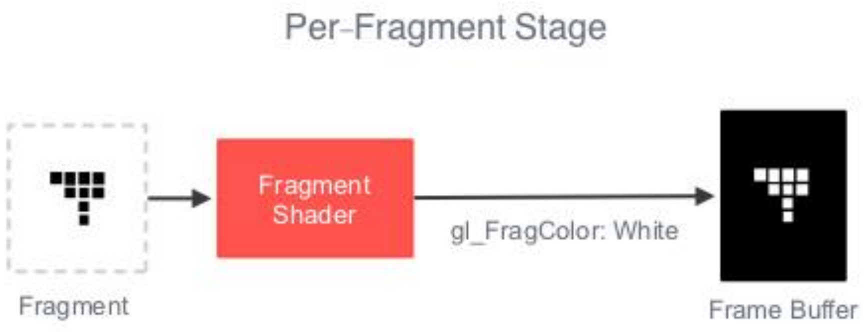
The main responsibility of a vertex shader is to compute the coordinates of a model into Clip Space. This calculation is placed in a required variable called gl_Position.



Usually, what is sent to a vertex shader from the CPU is not just the coordinates of a vertex, but also the normals of the vertex, color of the vertex, etc. These variables may be used in the Fragment Shader for lighting computations.

b) Fragment Shader

The fragment shader is the last operation in the OpenGL pipeline. It is where a pixel gets assigned a color. The basic operation of a fragment shader is to provide a color to each pixel. More specifically, the fragment shader takes the Uniform data and the output from the rasterizer and computes the color of the pixel for each fragment.



Inputs to a Fragment Shader

The fragment shader takes its input from the vertex shader. The fragment shader takes this interpolated data, along with Uniform data and computes a color to each pixel before they are sent to the framebuffer.

The fragment shader can receive:

Varying/ Out variable
Uniform data
Texture data

Outputs of a Fragment Shader

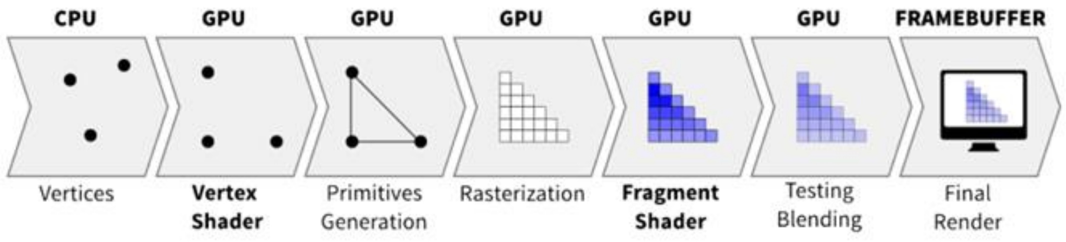
The fragment shader requires a vec4 as its output and this declares the color that should be assigned to each pixel.

For OpenGL 3.0 and beyond, we have to declare this variable as

out vec4 Fragment color;

The vertex and fragment shader programs are represented as C-style strings containing GLSL (Graphics Library Shading Language) language (vertexShaderSource and fragmentShaderSource) inside a regular C program that runs on the CPU.

2.1.6 OpenGL Rendering Pipeline



a) Vertex Specification

In Vertex Specification, an ordered list of vertices that define the boundaries of the primitive are listed. Along with this, one can define other vertex attributes like color, texture coordinates etc. Later this data is sent down and manipulated by the pipeline.

b) Vertex Shader

The vertex specification defined above now passes through Vertex Shader. Vertex Shader is a program written in GLSL that manipulates the vertex data. The ultimate goal of vertex shader is to calculate the final vertex position of each vertex. Vertex shaders are executed once for every vertex (in case of a triangle it will execute 3-times) that the GPU processes. The main job of a vertex shader is to calculate the final positions of the vertices in the scene.

c) Primitive Assembly

This stage collects the vertex data into an ordered sequence of simple primitives (lines, points or triangles). **Primitive** is constructed by connecting the vertices in a specified order.

d) Primitive Processing

Before the primitive is taken to the next stage, Clipping occurs. Any primitive that falls outside the view-volume, i.e. outside the screen, is clipped and ignored in the next stage.

e) Rasterization

What we ultimately see on a screen are pixels approximating the shape of a primitive. This approximation occurs in this stage. In this stage, both geometric and pixel data are converted into fragments. Each fragment square corresponds to a pixel in the framebuffer (i.e. screen). In this stage, pixels are tested to see if they are inside the primitive's perimeter. If they are not, they are discarded. If they are within the primitive, they are taken to the next stage. The set of pixels that passed the test is called a fragment. Color and depth values are assigned for each fragment square.

f) Fragment Shader

This user-written program in GLSL calculates the color of each fragment that the user sees on the screen. The fragment shader runs for each fragment in the geometry. The job of the fragment shader is to apply the final color or a texture for each pixel within the fragment.

g) Per-Fragment Operation

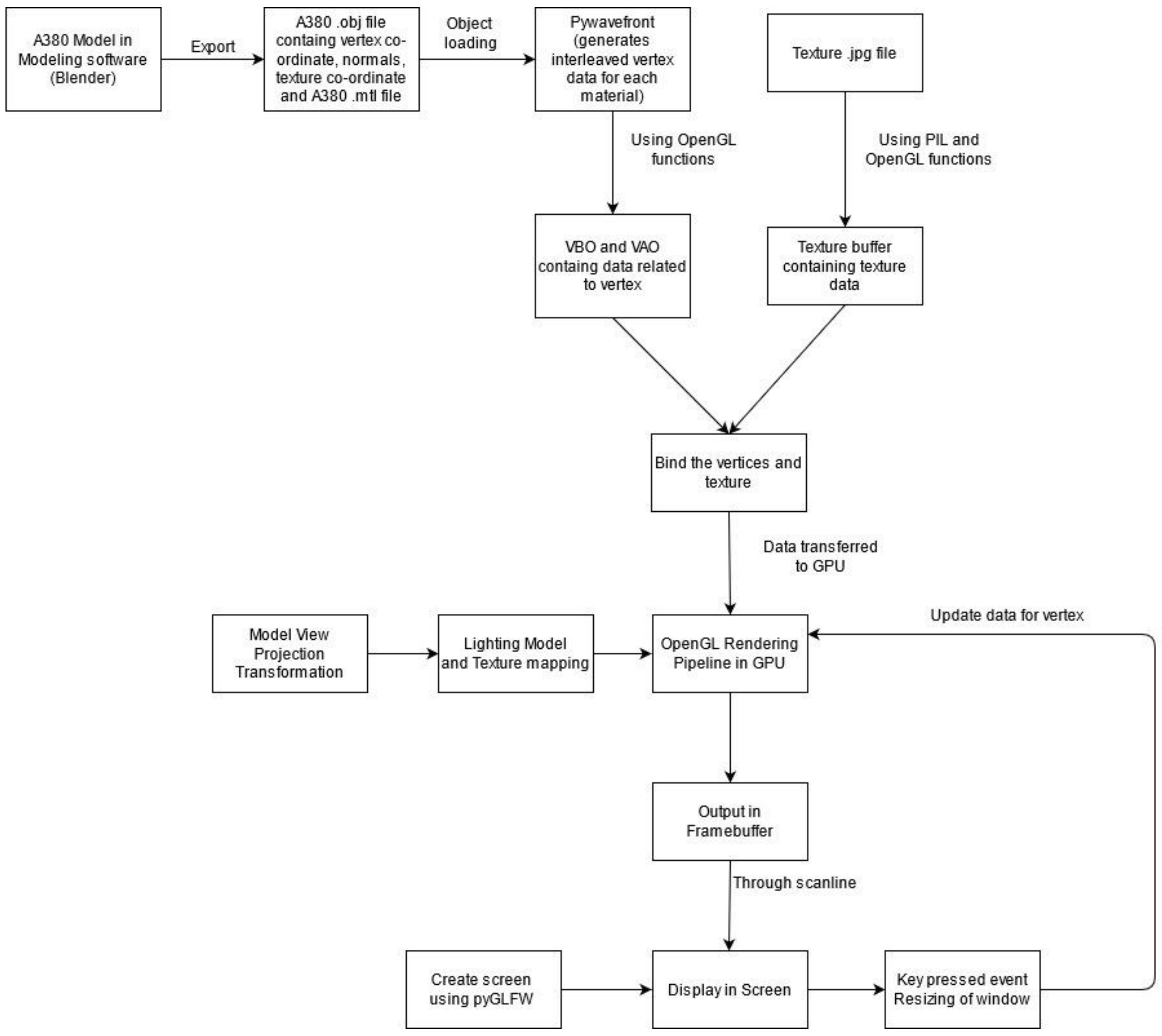
Before the pixels in the fragment are sent to the framebuffer, fragments are submitted to several tests like:

- Pixel Ownership test
- Scissor test
- Alpha test
- Stencil test
- Depth test

At the end of the pipeline, the pixels are saved in a Framebuffer, more specifically the Default-Framebuffer. These are the pixels that you see on the screen.

3 Methodologies

3.1 Block diagram


A380 Model in Modeling software (Blender)
Export
A380 .obj file containg vertex co-ordinate, normals, texture co-ordinate and A380 .mtl file
Object loading
Pywavefront (generates interleaved vertex data for each material)
Texture .jpg file
Using OpenGL functions
Using PIL and OpenGL functions
VBO and VAO containg data related to vertex
Texture buffer containing texture data
Bind the vertices and texture
Data transferred to GPU
Update data for vertex
Model View Projection Transformation
Lighting Model and Texture mapping
OpenGL Rendering Pipeline in GPU
Output in Framebuffer
Through scanline
Create screen using pyGLFW
Display in Screen
Key pressed event Resizing of window

3.2 Algorithms and Implementation

Translation

A translation process moves every point a constant distance in a specified direction. It can be described as a rigid motion. A translation can also be interpreted as the addition of a constant vector to every point, or as shifting the origin of the coordinate system.

If point (X, Y, Z) is to be translated by amount Dx, Dy and Dz to a new location (X', Y', Z') then new coordinates can be obtained by adding Dx to X, Dy to Y and Dz to Z as:

X' = Dx + X

Y' = Dy + Y

Z' = Dz + Z

The process can be represented in matrix as:

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & Tz \\ 0 & 0 & 0 & Ty \\ 0 & 0 & 0 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Scaling

Scaling is performed to resize the 3D-object that is the dimension of the object can be scaled(alter) in any of the x, y, z direction through S_x, S_y, S_z scaling factors.

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Rotation

Rotations in 3D are specified with an angle and a rotation axis. The angle specified will rotate the object along the rotation axis given. Using trigonometry it is possible to transform vectors to new rotated vectors given an angle. This is achieved through sine and cosine function.

A rotation matrix is defined for each unit axis in 3D space where the angle is represented as the theta symbol θ. θ is considered positive for counter-clockwise rotation and negative for clockwise direction.

Rotation along X-axis

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & 0 & -sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Rotation along Y-axis

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & 0 & -sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Rotation along Z-axis

$$\begin{bmatrix} X' \\ Y' \\ Z' \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & sin\theta & 0 & 0 \\ -sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

LookAt() is used to construct a viewing matrix where a camera is located at the eye position and looking at (or rotating to) the target point . The eye position and target are defined in world space. Camera's lookAt transformation consists of 2 transformations; translating matrix (**M$_T$**), and then rotating matrix (**M$_R$**).

For **Translation part**,

$$M_{\mathrm{T}} = \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For **Rotation part**, first compute the normalized forward vector from the target position **v$_t$** to the eye position **v$_e$** of the rotation matrix. Note that the forward vector is from the target to the eye position, not eye to target because the scene is actually rotated, not the camera.

Second, compute the normalized left vector by performing cross product with a given camera's up vector.

Finally, re-calculate the normalized up vector by doing cross product of the forward and left vectors. Note we do not normalize the up vector because the cross product of 2 perpendicular unit vectors also produces a unit vector.

These 3 basis vectors, , and are used to construct the rotation matrix **M$_R$** of lookAt, however, the rotation matrix must be inverted.

The rotation part **M$_R$** of lookAt is finding the rotation matrix from the target to the eye position, then inverting it. And, the inverse matrix is equal to its transpose matrix because it is orthogonal in which each column has unit length and is perpendicular to the other column.

$$M_R = \begin{pmatrix} l_x & u_x & f_x & 0 \\ l_y & u_y & f_y & 0 \\ l_z & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} l_x & u_x & f_x & 0 \\ l_y & u_y & f_y & 0 \\ l_z & u_z & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{T} = \begin{pmatrix} l_x & l_y & l_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Finally, the view matrix for camera's lookAt transform is:

$$M_{\text{view}} = M_R M_T = \begin{pmatrix} l_x & l_y & l_z & 0 \\ u_x & u_y & u_z & 0 \\ f_x & f_y & f_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} l_x & l_y & l_z & -l_x x_e - l_y y_e - l_z z_e \\ u_x & u_y & u_z & -u_x x_e - u_y y_e - u_z z_e \\ f_x & f_y & f_z & -f_x x_e - f_y y_e - f_z z_e \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

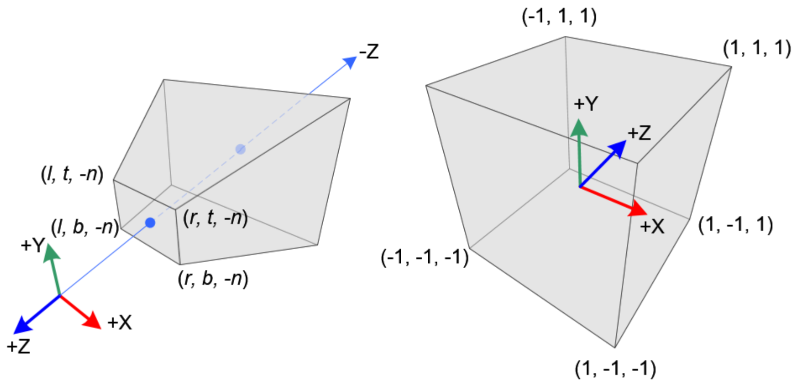
Create_Perspective_Projection()

In perspective projection, a 3D point in a truncated pyramid frustum (eye coordinates) is mapped to a cube (NDC);

the range of x-coordinate from [l, r] to [-1, 1],

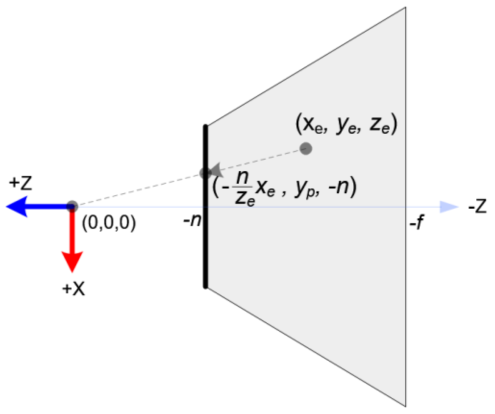
the y-coordinate from [b, t] to [-1, 1]

the z-coordinate from [-n, -f] to [-1, 1].

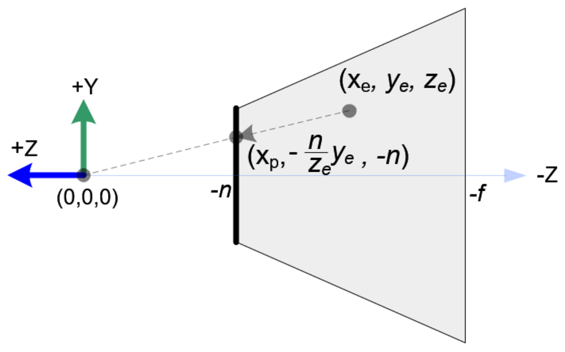


Perspective Frustum and Normalized Device Coordinates (NDC)

In OpenGL, a 3D point in eye space is projected onto the *near* plane (projection plane). The following diagrams show how a point (x_e, y_e, z_e) in eye space is projected to (x_p, y_p, z_p) on the *near plane.*



Top View of Frustum



Side View of Frustum

After some derivation we get the complete projection matrix as:

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

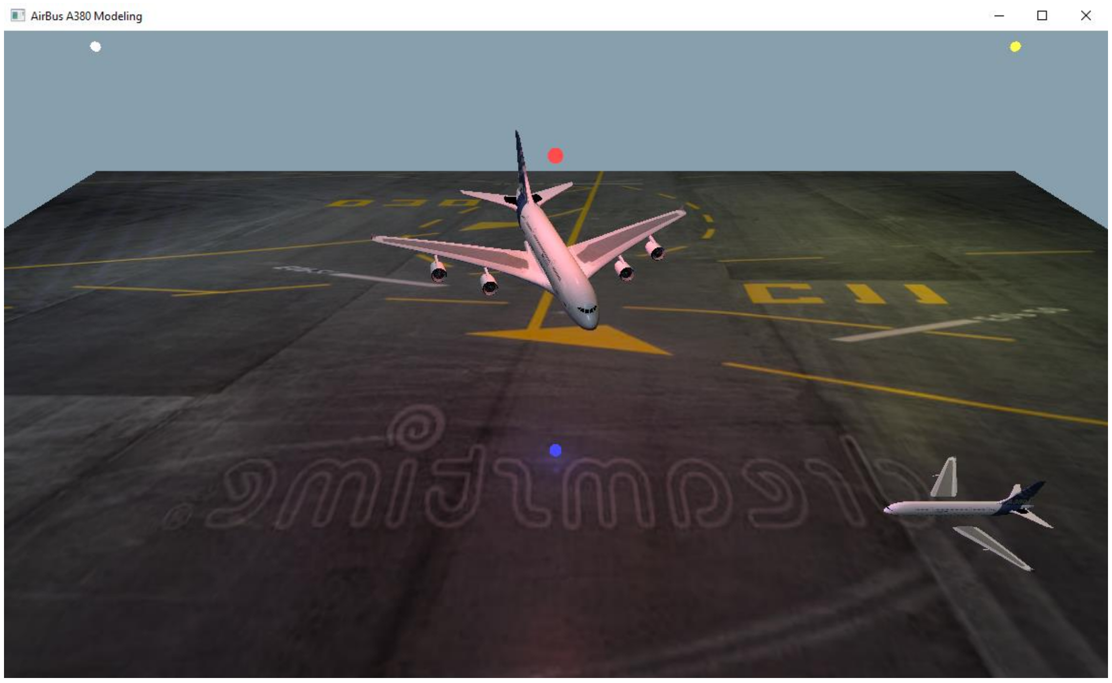
3.3 Languages and Tools Used

This project was implemented using Python 3.8. The object-oriented approach and its compatibility with OpenGL, GLFW as well as other libraries made it best fit in this project. We also used GLSL which is an OpenGL Shading Language.

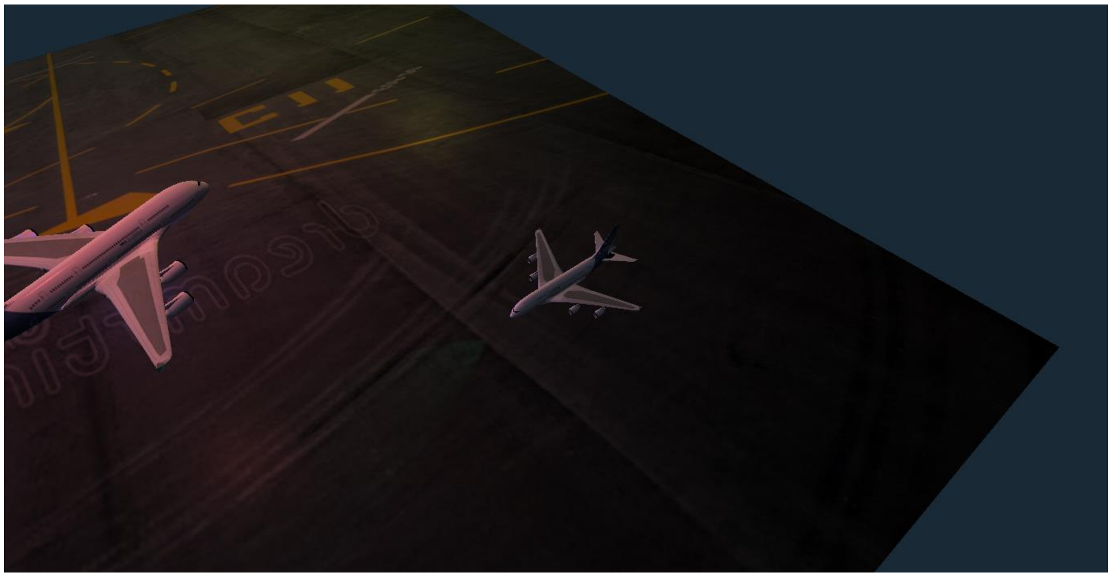
Some of the libraries used in this project were OpenGL, pyGLFW, PyWaveFront and PIL. pyGLFW was used to create a window and display the models as well as take inputs from the user. PyWaveFront was used to read the obj file and generate interleaved vertex data. PIL was used to load and manipulate texture image.

We used PyCharm was our IDE as it contained all the necessary libraries and had easy interface. Blender was also used to create the 3D model which was used I this project.

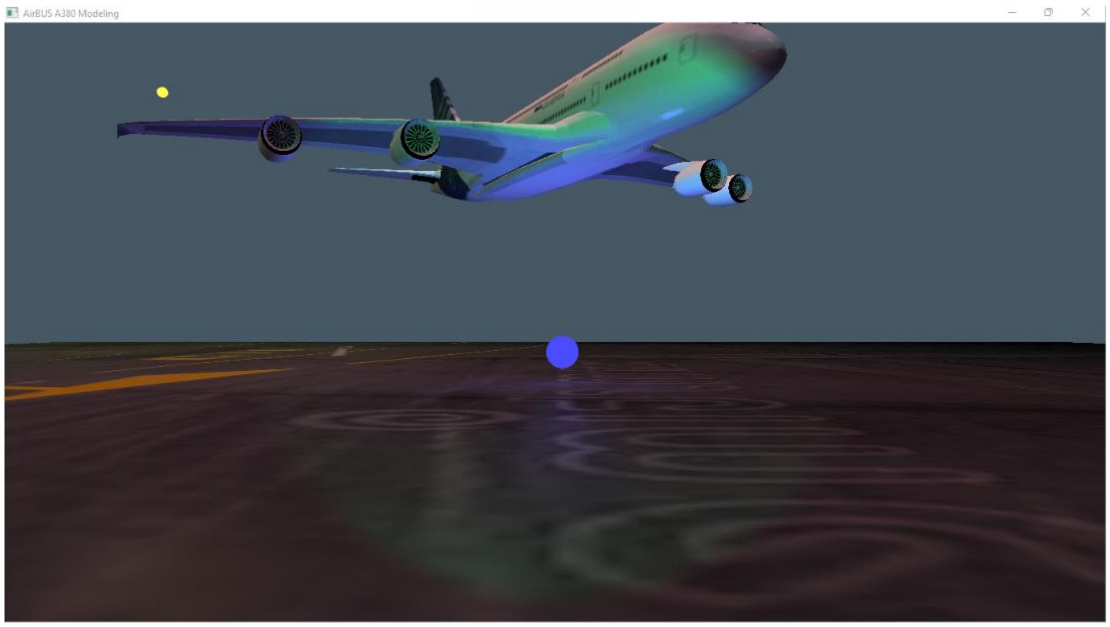
4 Result



Static Camera: Contains constantly movin center plane and manually controlled secondary plane



Following Camera: Focuses on the movement of secondary plane



Moving Camera: View secondary plane

5 Conclusion

We have successfully rendered the 3D model of A380 airbus using the necessary graphical algorithms. The model was rendered using python and doesn't require installation of any software. The airplane model was created using Blender and exported in obj format. The obj file was loaded and textures were linked using various OpenGL Libraries. Algorithms such as rotation, scaling, translation, lighting, shading, perspective view were implemented to create a realistic model of an airplane.

6 Limitations and Future Works

Although the model may seem flawless at a glance, it has some drawbacks. Some of the drawbacks are:

i. The scene cannot be controlled using mouse. This makes it harder to manually control the camera to view the object.
ii. With the lack of proper environment, the airplane model seems unrealistic.
iii. The model is not fully complete, lacking landing gears and other finer details.

This project is not limited to its current state. It can be improved expanding its use in various sectors. Some of the improvements are:

i. Additional inputs like that of mouse can be added for better navigation in the scene.
ii. The project can be made to be compatible with different file formats other than obj file
iii. It can be made to render the model that are input from the user
iv. The moving camera can be modified to give the experience of flying the plane

7 References

a. Computer Graphics ‘C version’ by Hearn & Baker

b. Computer Graphics: Principles and Practice in C by Foley et.al

c. https://github.com/pywavefront/PyWavefront

d. htps://www.blender.org/

e. https://learnopengl.com/

f. https://www.scratchapixel.com