Development of a 3D Modeling Software in C++

A Computer Graphics Project Report

Submitted By:

Saurav Adhikari (080BCT079)

Sujal Gyawali (080BCT087)

Upahar Khatiwada (080BCT094)

Utsav Dhakal (080BCT095)

Under the Guidance of:

Asst. Prof. Sanjivan Satyal

Pulchowk Campus, IOE

Department of Electronics and Computer Engineering

April 17, 2025

Abstract

This report presents the development of a 3D Modeling Software built using C++ and Raylib. The software provides a fundamental 3D modeling environment where users can load, view, and edit 3D objects with minimal dependencies. While Raylib was used solely for basic pixle rendering (DrawPiexl function) and some UI elements, all other core functionalities—including camera system, coordinate system, projection methods, surface detection, and model transformations—were implemented from scratch, following the principles taught in our computer graphics course.

The software supports OBJ file loading, allowing users to import 3D models and view them in Perspective or Orthographic projection modes. It offers two primary rendering styles: Wireframe Mode and Fill Mode (solid view). Additionally, users can modify vertex positions and export the modified models for further use.

This project emphasizes the mathematical foundations of 3D graphics, demonstrating the practical implementation of projection transformations, depth calculations, and rendering pipelines. By developing a functional yet lightweight modeling tool from first principles, this project provides valuable insights into the inner workings of 3D computer graphics systems without relying on high-level game engines.

Acknowledgment

We would like to express our sincere gratitude to our professors and mentors, whose guidance and support have been invaluable in the successful completion of this project. Their insights into computer graphics, mathematical foundations, and software development principles have significantly contributed to our understanding and implementation of this 3D modeling software.

We are especially grateful to Sanjivan sir for providing us with the opportunity to explore and implement graphics concepts from the ground up. Their encouragement and constructive feedback have been instrumental throughout our project development.

We also extend our appreciation to our classmates, whose discussions and shared experiences have helped us refine our ideas and overcome challenges. Lastly, we would like to acknowledge the open-source community, for helping us out whenever we were stuck on any part.

Without the support of these individuals, this project would not have been possible.

Contents

1

Introduction

1.1 Overview

3D modeling is a fundamental aspect of modern computer graphics, forming the backbone of applications in gaming, animation, simulation, CAD, and virtual reality. Most 3D modeling software relies on high-level libraries such as **OpenGL, DirectX, or Vulkan**, which abstract many complex mathematical operations. However, this project aims to build a minimalist yet functional 3D modeling software using **pure C++**, implementing most functionalities from scratch while using **Raylib only for rendering lines**.

The software allows users to:

- Load and view OBJ models.
- Toggle between Perspective and Orthographic projections.
- Switch between Wireframe and Fill modes.
- Modify vertex positions of models
- Export the modified model for further use.

By constructing fundamental components—such as **camera transformations, projection matrices, and surface detection algorithms**—without relying on external 3D engines, this project serves as a **learning-oriented exploration of 3D graphics programming**.

1.2 Objectives

The primary objectives of this project are:

1. To develop a simple 3D modeling tool using C++ with minimal external dependencies.
2. To implement fundamental graphics transformations manually, including coordinate systems, projections, and model manipulation.
3. To support OBJ file loading, enabling users to import 3D models.
4. To allow both Perspective and Orthographic views for different visualization needs.
5. To enable vertex-level editing and export modified models.
6. To reinforce concepts learned in class, such as projection transformations, rendering techniques, and camera control.

1.3 Scope

This project aims to provide a **basic yet functional** 3D modeling environment while staying within certain limitations:

Included Features:

- OBJ file loading and visualization.
- Perspective and Orthographic projection support.
- Wireframe and Fill rendering modes.
- Basic vertex manipulation (editing vertex positions).
- Model export functionality.

Excluded Features:

- Advanced rendering techniques (e.g., shading, textures)
- Animation support.

- Subdivision surface, Extruding, Smooth shading etc.
- Load multiple OBJ files into a single instance

The project serves as a foundational exercise in graphics programming, focusing on manual implementation of core principles rather than advanced rendering techniques.

2

Background Theory

2.1 Computer Graphics Concepts

Computer graphics is the field of generating and manipulating visual content using mathematical models and algorithms. The core concepts include:

- **Rasterization:** The process of converting geometric data into pixels for display on a screen.
- **Vector Graphics:** Using mathematical equations to define shapes, enabling scalable and precise representations.
- **3D Transformations:** Operations such as translation, rotation, and scaling used to manipulate objects in a 3D space.
- **Rendering Pipeline:** The sequence of steps from 3D model processing to final image generation.

2.2 3D Coordinate Systems

In a 3D space, objects are represented using a coordinate system consisting of three perpendicular axes: **X, Y, and Z**. Each point in 3D space is defined as (x, y, z), where:

- The **X-axis** typically represents the horizontal direction.
- The **Y-axis** represents the vertical direction.

- The **Z-axis** represents depth (into or out of the screen).

Lines and objects are constructed from these points. A line is defined by two points (x_1, y_1, z_1) and (x_2, y_2, z_2), while complex objects are made up of multiple connected vertices forming edges and faces.

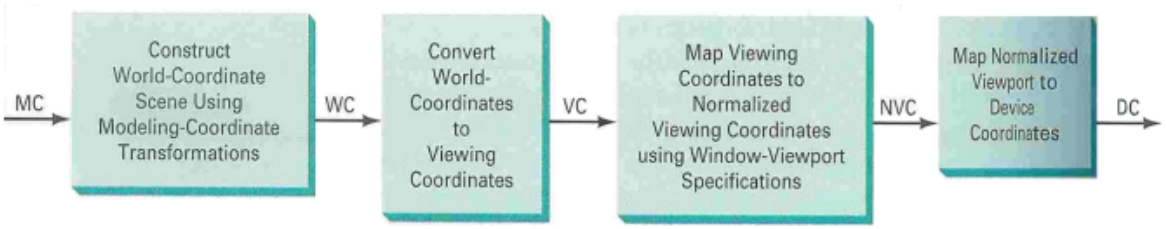
Coordinate transformations such as **translation, rotation, and scaling** allow for object manipulation within the 3D environment. These transformations are typically represented using **homogeneous coordinates** and matrix operations to efficiently modify object positions and orientations. A point (x,y,z) in 3D homogenous system can be represented as:

$$\mathbf{P} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Here, w is for perspective representation such that, x' = x/w and so on. Usually for orthographic and 2D systems it is taken as 1

2.3 3D Viewing Pipeline

The **3D viewing pipeline** is the sequence of transformations that convert a 3D scene from **model coordinates** to **device coordinates** for rendering on the screen. This process ensures that the objects are positioned correctly, projected accurately, and mapped to the display. The pipeline consists of the following major stages:



2.3.1 Modeling Coordinate (MC) System to World Coordinate (WC) System

- Objects in a 3D scene are first defined in their own **local coordinate system** (modeling coordinates).

- Using **modeling transformations**, these objects are positioned within a **global coordinate system**, known as the **world coordinate system** (WC).
- This step allows multiple models to coexist in the same 3D space with different positions and orientations.

2.3.2 World Coordinates (WC) to Viewing Coordinates (VC)

- In this stage, the **world coordinate system** is transformed into the **viewing coordinate system**.
- The **viewing transformation (camera transformation)** defines a **camera or observer's perspective** by positioning and orienting the camera.
- This transformation involves:
  - Defining the **camera's position** in the scene.
  - Orienting the **viewing direction**.
  - Establishing the **up direction**.
- The result is a new coordinate system centered around the camera, where the **camera is at the origin**.

2.3.3 Viewing Coordinates (VC) to Normalized Viewing Coordinates (NVC)

- After the scene is transformed into the camera's view space, it must be projected onto a 2D plane.
- This involves defining the **viewing volume**, which is a **frustum** for perspective projection or a **rectangular box** for orthographic projection.
- The **window-to-viewport transformation** maps the selected portion of the scene into a **normalized coordinate system**, where:
  - The viewing volume is mapped into a standard cube (for 3D scenes) or a square (for 2D scenes).

  - This normalization step ensures uniformity, allowing scenes to be projected onto different screen resolutions while preserving their aspect ratio.

2.3.4 Normalized Viewing Coordinates (NVC) to Device Coordinates (DC)

- Finally, the **normalized coordinates** are mapped onto **actual device coordinates**.
- This step involves:
  - Mapping the **normalized viewport coordinates** to **screen coordinates (pixels)**.
  - Applying any necessary **clipping** to remove parts of objects outside the viewport.
  - Rasterizing the projected 2D scene onto the display device.

Conclusion

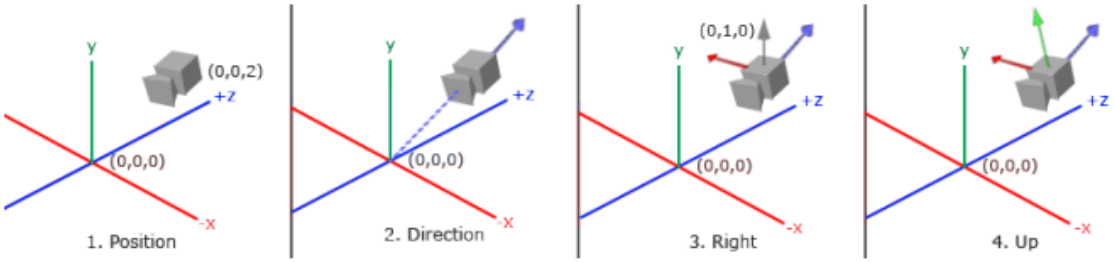
The 3D viewing pipeline ensures that objects in a 3D scene go through a structured transformation process, ultimately rendering them onto a 2D screen in a visually meaningful way. Understanding these stages helps in implementing **custom rendering engines** and working with **graphics APIs** like OpenGL or DirectX.

2.4 Viewing Coordinate Transformation

In 3D graphics, the viewing / camera coordinate system represents all vertex coordinates relative to the observer's or say camera's perspective, with the camera or the eye positioned at the origin. The transformation from world space to view space is achieved using the view matrix, which adjusts all object coordinates based on the camera's position and orientation.

To define a camera, we require its position in world space, the direction it is looking at, a rightward-pointing vector, and an upward-pointing vector. These three vectors form an orthonormal basis, creating a local coordinate system where:

- The *forward* vector **w** is the normalized inverse of the viewing direction.
- The *right* vector **u** is computed as the cross product of a global up vector and the forward vector.
- The *up* vector **v** is obtained as the cross product of the forward and right vectors.



Thus, the camera coordinate system is established using three mutually perpendicular unit vectors. Next, we develop a transformation matrix in the following way to transform from World View to Camera View.

$$R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{\text{camera}} = R \cdot T = \begin{bmatrix} u_x & u_y & u_z & -(u_x t_x + u_y t_y + u_z t_z) \\ v_x & v_y & v_z & -(v_x t_x + v_y t_y + v_z t_z) \\ w_x & w_y & w_z & -(w_x t_x + w_y t_y + w_z t_z) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

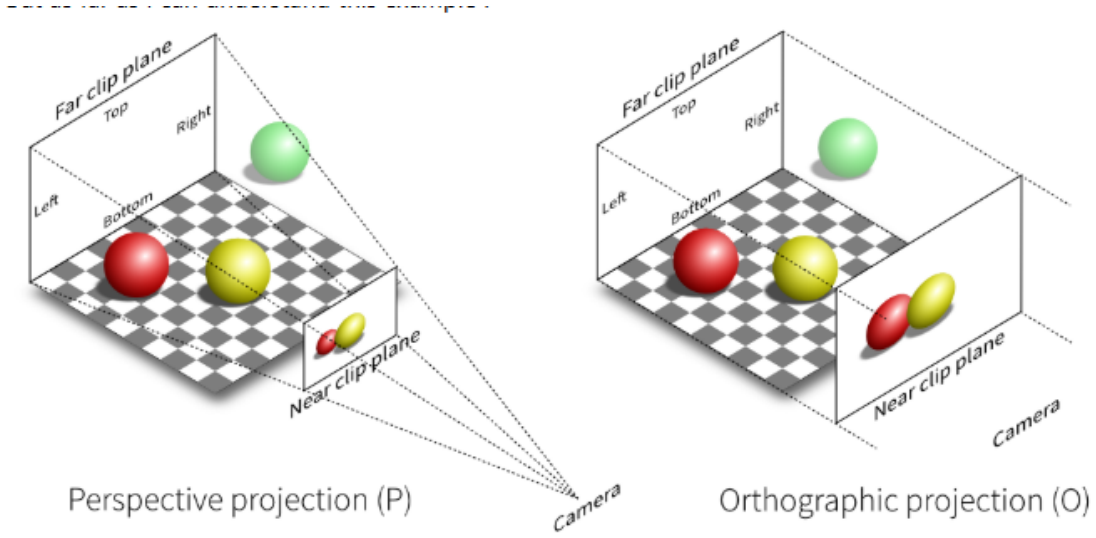
$$P_c = M_{\text{camera}} \cdot P = \begin{bmatrix} u_x & u_y & u_z & -(u_x t_x + u_y t_y + u_z t_z) \\ v_x & v_y & v_z & -(v_x t_x + v_y t_y + v_z t_z) \\ w_x & w_y & w_z & -(w_x t_x + w_y t_y + w_z t_z) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

Note: T is the camera's position in our world

Here P is the world cordinate and finally Pc is the cordinate on the basis of the cordinate system defined based on how the camera **SEES** our world.

2.5 Projection

Projection is the process of mapping 3D points onto a 2D plane. Two commonly used projection techniques in computer graphics are orthographic projection and perspective projection.



2.5.1 Orthographic Projection

Orthographic projection maintains parallel lines and does not account for depth perception. This projection is often used in engineering and architectural applications where accurate dimensions are required.

An orthographic projection frustum is defined by six parameters:

- l, r: Left and right clipping planes
- b, t: Bottom and top clipping planes
- n, f: Near and far clipping planes

The transformation matrix for an orthographic projection is given by:

$$P_o = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

This matrix ensures that coordinates are scaled and translated to fit within a normalized device coordinate (NDC) range of $[-1, 1]$.

2.5.2 Perspective Projection

Perspective projection mimics how the human eye perceives depth. Objects further away appear smaller, giving a more realistic view.

A perspective frustum is defined similarly to the orthographic case, but the projection includes a division by the depth value (w-coordinate). The perspective projection matrix is:

$$P_p = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{2.2}$$

After transformation, each point's coordinates are divided by the homogeneous coordinate w to map them into normalized device coordinates (NDC).

2.5.3 Normalized Device Coordinates (NDC)

After applying the projection matrix, points are mapped into a normalized coordinate space where:

- x, y, z values lie in the range $[-1, 1]$
- The Z-depth helps with depth-buffering and occlusion handling

This transformation is crucial because it standardizes geometry for subsequent rendering operations.

2.5.4 Transformation from Camera Space to NDC

The complete transformation from camera space to NDC involves multiplying the camera transformation matrix P_c with the projection matrix (P_o or P_p). This can be expressed as:

$$P_{final} = P \cdot P_c \tag{2.3}$$

where P represents either P_o or P_p.

Thus, a vertex v in camera space is transformed as:

$$v' = P_{final} \cdot v \tag{2.4}$$

This final transformation ensures the correct mapping from world coordinates to screen space, ready for rasterization.

Clipping: Also in this stage, clipping can also be performed to ensure that the transformed values are within the range of $[-1, 1]$. For this, complex algorithms like Sutherland-Hogdman along with several other ideas exist. In clipping stage we need to make sure that the faces or the vertices completely outside is not rendered, ones that are completely inside is rendered while for the ones which are partially inside and partially outside the

2.5.5 NDC to Device Cordinates Using View Port Transformation

Finally the NDC are converted to device coordinates based on the devices height and width which are specific to individual devices. This ensures that no matter where the scene was constructed we can rasterize it into any capable device i.e our NDC becomes independent of hardware which is a crucial factor of graphics programming.

$$P_d = \begin{bmatrix} x_d \\ y_d \end{bmatrix} = \begin{bmatrix} \frac{x_{\max} - x_{\min}}{2} x_{ndc} + \frac{x_{\max} + x_{\min}}{2} \\ \frac{y_{\max} - y_{\min}}{2} y_{ndc} + \frac{y_{\max} + y_{\min}}{2} \end{bmatrix}$$

where Xmin, Ymin, Xmax, Ymax are the values of the device screen features.

2.6 Visible Surface Detection

In computer graphics, **visible surface detection** is a crucial technique used to determine which surfaces of 3D objects should be rendered and displayed on the screen. Since a 3D

scene often contains multiple overlapping objects, it is essential to identify and eliminate hidden surfaces to create an accurate and realistic representation. Without proper visible surface detection, objects behind other objects might still be drawn, leading to incorrect visual results and wasted computational resources.

Several algorithms exist to handle this problem, each with its strengths and limitations. The choice of algorithm depends on factors such as scene complexity, performance constraints, and rendering pipeline requirements. Two commonly used methods for visible surface detection are **Back-Face Culling** and the **Painter's Algorithm**.

2.6.1 Back-Face Culling

Back-face culling is a technique used to optimize rendering by removing surfaces that are not visible to the camera. In a closed 3D object, half of the surfaces face outward, while the other half face inward. Since the inward-facing surfaces are never visible from an external viewpoint, rendering them is unnecessary and wasteful.

This technique works by examining the orientation of each triangle's normal vector relative to the viewer's position. If the normal vector points away from the camera, the triangle is considered to be facing away and is culled (discarded). The decision is typically made using the **dot product** between the view direction and the triangle's normal vector:

$$\mathbf{N} \cdot \mathbf{V} < 0$$

where $\mathbf{N}$ is the normal vector of the triangle and $\mathbf{V}$ is the view vector. If the result is negative, the surface is facing away and can be ignored. This method is widely used in real-time rendering applications such as video games and simulations, as it significantly reduces the number of polygons that need to be processed.

2.6.2 Painter's Algorithm

The **Painter's Algorithm**, introduced by Hewells in 1972, is a depth-based technique that sorts and renders surfaces from the farthest to the nearest, similar to how an artist paints a scene—starting with the background and progressively adding objects in the foreground.

The algorithm follows these steps:

1. **Sort the surfaces** based on their depth, from farthest to nearest (sorted by their z-values).
2. **Scan and convert surfaces** onto the 2D screen in the sorted order, rendering the farthest first.
3. **Check for overlapping surfaces** and resolve visibility conflicts using bounding box comparisons or depth-based tests.
4. **Update the frame buffer** with the correct color and shading information for each pixel.
5. **Repeat the process** for all surfaces in the scene.

If two surfaces overlap, additional depth comparisons (such as the **mini-max method**) are performed to determine which surface should be displayed. However, a major limitation of this approach is handling **cycles**, where three or more overlapping surfaces form an unsortable loop, requiring additional splitting operations to resolve visibility.

2.6.3 Combining Techniques

- **Back-face culling** eliminates more than half of the back-facing surfaces, significantly reducing the rendering load.
- **Painter's algorithm** ensures that any remaining hidden surfaces are not rendered, producing the correct visible output.

Visible surface detection plays a key role in modern rendering pipelines, often in combination with **Z-buffering**, a more advanced depth-based technique that stores depth values for each pixel to dynamically resolve visibility. Choosing the right approach depends on the specific requirements of the application, such as whether real-time performance or accuracy is more critical.

2.7 OBJ File Format

The OBJ file format is a widely used, simple, and open file format for representing 3D geometry. It was developed by Wavefront Technologies and is commonly used for storing 3D models, including vertices, faces, normals, texture coordinates, and material properties. The format is human-readable and stores geometric data as plain text, making it easy to parse and manipulate.

An OBJ file primarily consists of the following elements:

- **Vertices (v)**: Each vertex is represented by three coordinates (x, y, z).
- **Texture Coordinates (vt)**: These specify how textures map onto the model's surface.
- **Normals (vn)**: Define the orientation of a surface for shading purposes.
- **Faces (f)**: Defined by indices referencing the vertices, normals, and texture coordinates.
- **Material Files (mtllib)**: A separate MTL file may be referenced to define materials, colors, and textures.

A sample OBJ file is shown below:

```
# Sample Cube OBJ File
v 0.0 0.0 0.0
v 1.0 0.0 0.0
v 1.0 1.0 0.0
v 0.0 1.0 0.0
vn 0.0 0.0 1.0
vt 0.0 0.0
vt 1.0 0.0
vt 1.0 1.0
vt 0.0 1.0
f 1/1/1 2/2/1 3/3/1
f 1/1/1 3/3/1 4/4/1
```

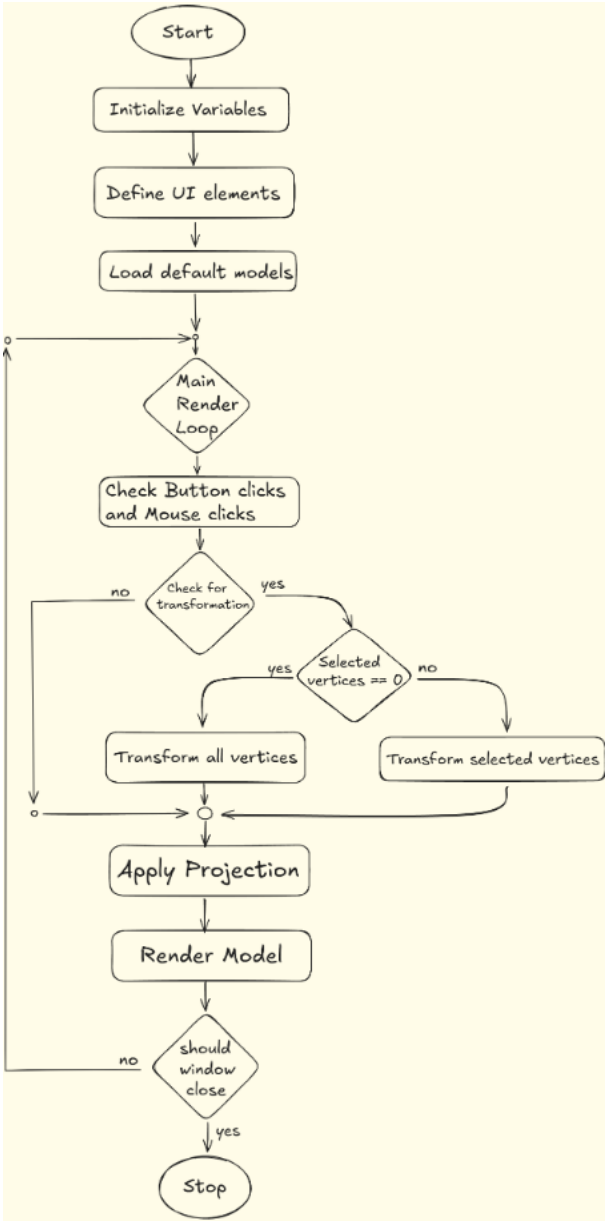
In this example:

- Four vertices (v) define the 3D positions of the model.
- A normal vector (vn) is specified for shading.
- Texture coordinates (vt) map a texture to the object.
- Faces (f) reference vertex, texture, and normal indices.

OBJ files are widely used due to their simplicity, but they do not support advanced features like animation or complex shading directly. For rendering, the model needs to be parsed and converted into a format that can be processed by the graphics pipeline.

3

Flowchart


Start
Initialize Variables
Define UI elements
Load default models
Main Render Loop
Check Button clicks and Mouse clicks
Check for transformation
no
yes
Selected vertices == 0
yes
no
Transform all vertices
Transform selected vertices
Apply Projection
Render Model
should window close
no
yes
Stop

4

Implementation

4.1 Code Structure

4.1.1 Core Classes and Structures

Structure: Coordinate

The `coordinate` struct represents a 4D point in homogeneous coordinates, denoted as (x, y, z, w). These coordinates are essential for transformations and projections in computer graphics and geometry. By introducing an additional coordinate w, homogeneous coordinates enable efficient matrix operations, including translation, rotation, and scaling. This unified representation simplifies complex transformations and allows for seamless manipulation of geometric objects in 3D space. Homogeneous coordinates are widely used in applications such as computer graphics, 3D modeling, and computer vision to enhance computational efficiency and flexibility in transformations.

Class: normVector

- Represents a 3D vector with components (x, y, z).
- **Key Methods:**
  - **Normalization**: Scales the vector to unit length using the formula:

$$x' = \frac{x}{\text{Magnitude}}, \quad y' = \frac{y}{\text{Magnitude}}, \quad z' = \frac{z}{\text{Magnitude}}$$

where

$$\text{Magnitude} = \sqrt{x^2 + y^2 + z^2}$$

  - **Cross Product**: Computes the perpendicular vector to two vectors using:

$$\mathbf{A} \times \mathbf{B} = \begin{bmatrix} yv_z - zv_y \\ zv_x - xv_z \\ xv_y - yv_x \end{bmatrix}$$

  - **Magnitude**: Calculates the length of the vector using:

$$\text{Magnitude} = \sqrt{x^2 + y^2 + z^2}$$

- Used for calculating normals, camera axes, and transformations.

Class: Matrix

- Represents a 4×4 transformation matrix commonly used in 3D graphics to perform operations such as translation, scaling, and rotation on objects in 3D space.
- **Key Features:**
  - **Identity Matrix**: The matrix is initialized as an identity matrix, which means it doesn't alter any object it is multiplied with. The identity matrix has ones along the diagonal and zeros elsewhere, signifying no transformation.
  - **Translation**: This matrix operation modifies the position of an object in 3D space. The translation is stored in the last column of the matrix, where the values of xx, yy, and zz represent how much the object should be moved along each axis.
  - **Scaling**: Scaling resizes objects, either uniformly or non-uniformly, by adjusting the diagonal elements of the matrix. Scaling along the xx, yy, or zz axes is controlled by setting the corresponding values along the diagonal to desired scale factors.
  - **Rotation**: Rotation is handled using trigonometric functions, where each axis (X, Y, or Z) has its own transformation matrix. The angle of rotation is given as a parameter, and the matrix is filled with cosine and sine values based on the angle to rotate the object.
  - **Matrix Multiplication**: The class supports multiplying two matrices together. This operation is essential in combining multiple transformations, as

the order of matrix multiplication determines the resulting transformation. It also supports multiplying the matrix with a coordinate to transform that point.

  - **Transpose**: The transpose operation swaps the rows and columns of the matrix. This is especially useful for camera transformations, where the view matrix requires transposition for correct calculations.

- **Constructor**: The constructor allows creating a matrix based on the desired transformation type. It can generate an identity matrix or apply translation, scaling, or rotation based on input parameters.

- **Rotation with Basis Vectors**: The matrix also supports defining rotations using three orthonormal vectors, which can be used for custom rotation axes. The vectors represent the new orientation of the object in 3D space, allowing for more complex transformations.

Class: Obj

- The `obj` class is designed to represent a 3D object loaded from an OBJ file, which is a popular format for 3D models.

- **Key Data Members:**

  - **Vertices**: This stores both the original and transformed positions of the vertices in 3D space. The original vertices are loaded from the OBJ file, and the transformed vertices are the result of applying transformations such as scaling, translation, or rotation.

  - **Normals**: Normals are vectors that are perpendicular to the surfaces of the object. They are essential for lighting calculations and for determining which faces are visible (back-face culling).

  - **Faces**: These are the triangles or polygons of the 3D object. The faces are defined by indices that refer to the vertices array. Each triangle consists of three vertex indices.

  - **Selected Vertices**: This tracks which vertices have been selected for editing. This feature is typically used in 3D modeling software to allow users to

manipulate specific parts of the object.

- **Key Methods:**
  - `loadObject()`: This method is responsible for parsing the OBJ file and loading the vertices, normals, and faces into their respective data structures.
  - `applyTransform()`: This method applies transformation matrices to the vertices, allowing for operations such as scaling, rotating, or translating the object.
  - `applyProjection()`: This method projects the 3D vertices into 2D screen space, which is necessary for rendering the object on a 2D display.
  - `drawObject()`: This method renders the object, typically using either scanline filling (for solid polygons) or wireframe rendering (for outline-based rendering).
  - `calculateNormals()`: This method computes the face normals for shading and back-face culling. The normals are essential for proper lighting calculations.
  - `selectVertices()`: This method allows the selection of specific vertices, typically for editing or transforming purposes.
  - `transformSelectedVertex()`: This method applies transformations (such as rotation or scaling) to the selected vertices.

4.2 Key Functions

4.2.1 scanLineFill()

- Fills a triangle using the scanline algorithm.
- **Steps:**
  1. Determine the minimum and maximum Y-coordinates of the triangle.
  2. For each scanline (Y-coordinate), compute the left and right X-boundaries.
  3. Fill pixels between the left and right boundaries.
- Used for rendering solid triangles with flat shading.

4.2.2 Bresenham Line Drawing Algorithm

Bresenham's line drawing algorithm is an efficient method for rasterizing straight lines on a pixel grid without using floating-point arithmetic. It works by calculating the differences Δx and Δy between the start and end points of the line and then determining a decision parameter p to decide whether the next pixel should be placed directly in line or diagonally. This decision parameter is updated iteratively, ensuring the most accurate approximation of the line using only integer calculations. The algorithm is widely used in computer graphics for rendering straight lines with high performance, making it an essential technique in 2D graphics applications.

4.2.3 main() Function

- The main loop handles:
  - **UI Rendering**: Buttons for object selection, mode toggles, and file operations. The interface provides an intuitive way to manage objects and modify scene properties efficiently.
  - **Input Handling**: Mouse and keyboard interactions allow users to manipulate objects and navigate the scene dynamically. Keybindings enable quick access to transformation modes, selection tools, and camera adjustments.
  - **Camera Control**: Orbiting, zooming, and view switching allow for flexible scene exploration. The camera adjusts based on user inputs, ensuring smooth navigation in both perspective and orthographic projections.
  - **Object Transformation**: Rotation, translation, and scaling are applied to objects or selected vertices. These transformations are performed based on user inputs, allowing for precise modifications to the scene.
  - **Rendering**: Draws objects, axes, and UI elements to maintain an interactive visual representation. The rendering pipeline ensures smooth frame updates and proper scene visualization in real time.

4.3 Key Handling Logic

4.3.1 Camera Controls

- **Left-Click Drag**: Orbits the camera around the object.
  - Computes camera axes (u, v, w) based on the current view direction.
  - Applies rotation matrices to update the camera position.
- **Mouse Wheel**: Zooms in/out by adjusting the camera distance.
- **Keys 1-3**: Switches to orthographic views (front, top, side).
- **Key 5**: Toggles between perspective and orthographic projection.

4.3.2 Object Transformation

- **Rotation:**
  - Press `R` to enable rotation mode.
  - Use `X`, `Y`, or `Z` to select the rotation axis.
  - Drag the mouse to rotate the object or selected vertices.
- **Translation:**
  - Press `T` to enable translation mode.
  - Use `X`, `Y`, or `Z` to select the translation axis.
  - Drag the mouse to move the object or selected vertices.
- **Scaling:**
  - Press `S` to enable scaling mode.
  - Drag the mouse to scale the object or selected vertices.
- **Vertex Selection:**
  - Right-click to select vertices in edit mode.
  - Press `Q` to clear the selection.

4.4 UI Interactions

4.4.1 Buttons

- Load primitives (cube, teapot, sphere).
- Toggle modes (edit, view, wireframe).
- Import/export OBJ files.

5

Results and Discussion

5.1 Challenges Faced and Solutions

During the development of this project, we encountered numerous challenges that required careful debugging and problem-solving. One of the initial difficult aspects was implementing the camera system correctly. Initially, we faced issues with the order of normal vectors, which led to distorted results. After identifying the mistake, we corrected the normal vector calculations to ensure proper rendering of the 3D objects.

Similarly, when implementing projections, both orthographic and perspective projections resulted in flipped images. This issue arose due to incorrect transformations applied during the projection matrix computations. After analyzing the transformation pipeline, we adjusted the matrices to correctly map the 3D coordinates to the 2D screen space.

Visible Surface Detection was one of the most challenging features to implement. Back-face culling worked efficiently for simple objects like cubes and spheres, successfully eliminating over half of the back-facing polygons. However, when applied to complex models such as the teapot, the method alone was insufficient, as some hidden faces remained visible due to the nature of the model's geometry. To address this, we implemented the Painter's Algorithm, which sorts and renders polygons based on their depth. Although it is not as efficient as the depth buffer method, it provided a functional solution for ensuring that hidden surfaces were not rendered incorrectly.

Another major challenge arose after modifying an object's geometry, as the normal vectors needed to be recalculated. If left unchanged, this led to incorrect shading and distorted rendering in view mode. To fix this, we implemented a mechanism to manually

recompute or transform normals dynamically based on the modifications made to the object.

Additionally, implementing features such as zooming and toggling between orthographic and perspective views introduced inconsistencies. To address this, we established a standardized approach for defining parameters in both orthographic and perspective view volumes, ensuring a seamless transition between the two.

While handling OBJ file imports, we encountered different types of files—some containing normals but lacking textures, some with textures but no normals, and others with both or neither. To account for all variations, we developed the loadObject function, which dynamically adapts to the structure of the input file, ensuring robust and flexible loading of 3D models.

Overall, the project provided valuable insights into 3D rendering, visibility determination, and object manipulation. The implementation challenges helped solidify our understanding of projection techniques, surface detection, and transformation matrices, making this a highly educational experience.

5.2 Screenshots

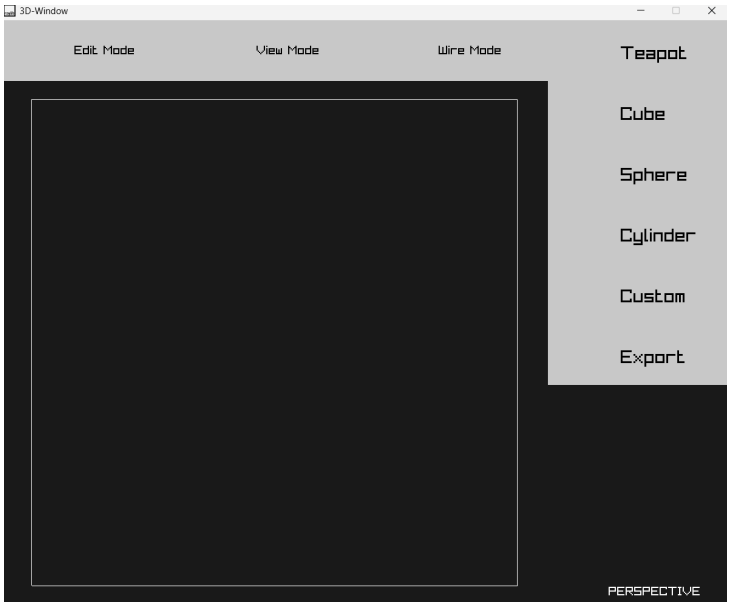


Figure 5.1: Basic UI Design of the Engine

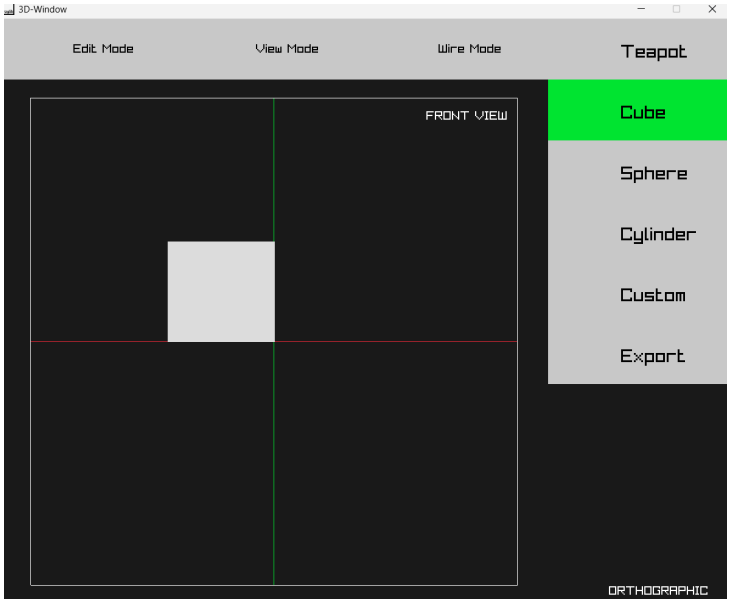


Figure 5.2: Orthographic Projection showcasing the Front View

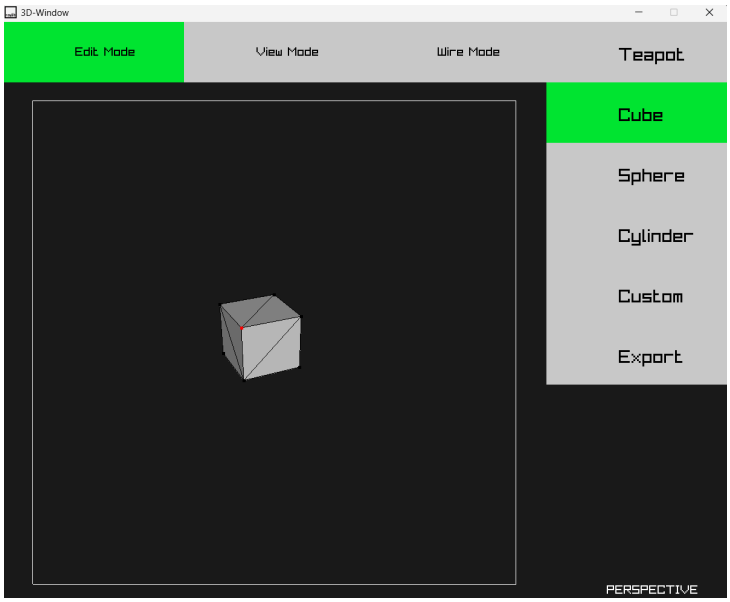


Figure 5.3: Perspective Projection showcasing the Edit Mode

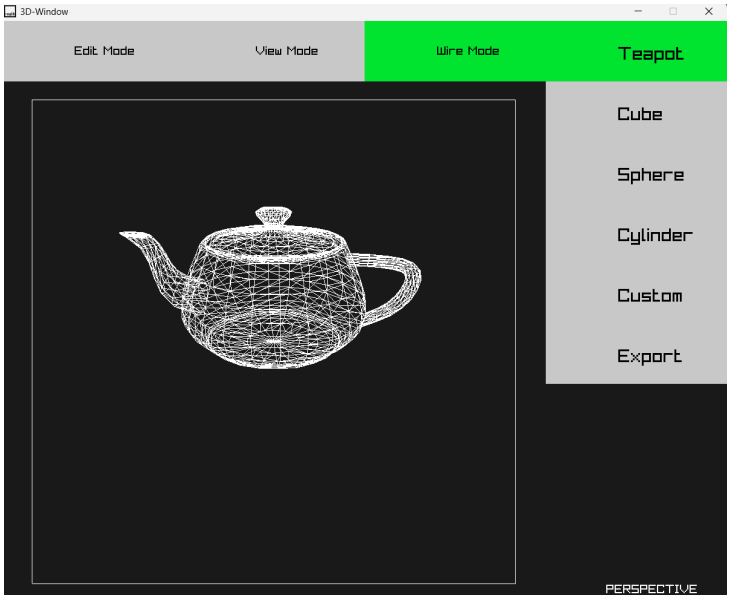


Figure 5.4: Perspective Projection showcasing the Wire Mode

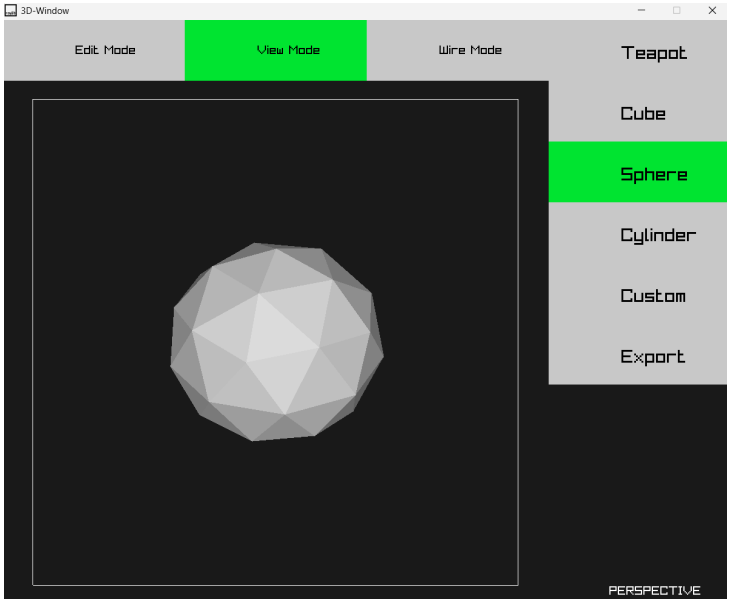


Figure 5.5: Perspective Projection showcasing the View Mode

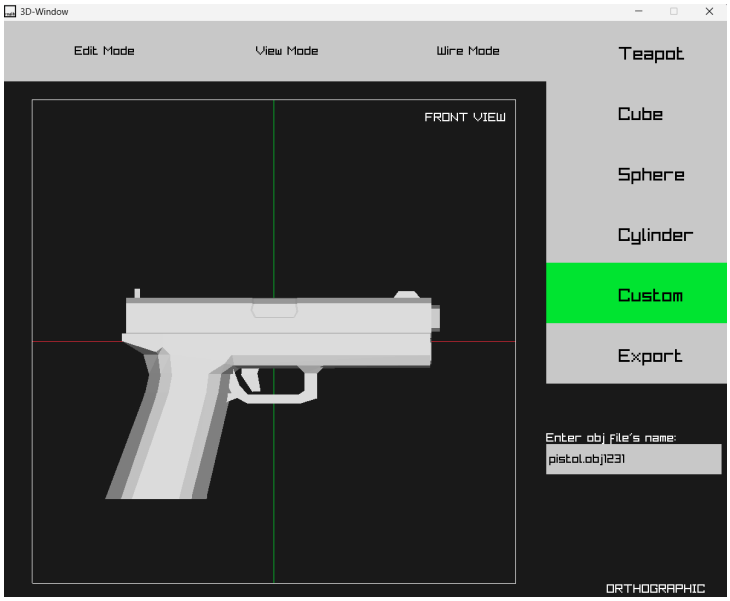


Figure 5.6: Loading a Custom obj file

6

Conclusion and Future Work

6.1 Limitations

- **Basic Lighting and No Texturing:** Only basic diffuse lighting is implemented, and there is no support for textures or advanced lighting effects like shadows or specular highlights.
- **Supports Only Triangular Faces:** The program cannot handle .obj files with quad faces or other polygons, limiting its compatibility with more complex models.
- **No Proper Clipping:** Clipping is handled by clamping vertex coordinates, which is not a robust solution. Proper clipping algorithms are missing.
- **No Depth Buffer (Z-Buffering):** The program relies on sorting faces by their Z-coordinates instead of using a depth buffer, leading to potential rendering artifacts.
- **Limited Camera and Object Manipulation:** Camera controls are basic, and object manipulation (translation, rotation, scaling) is global, with no support for fine-grained editing of individual vertices or faces.

6.2 Future Enhancements

- **Implement Z-Buffering:** Add a depth buffer to handle hidden surface removal accurately and eliminate rendering artifacts.

- **Add Support for Textures and Materials:** Parse and use .mtl files and textures to enhance the visual quality of rendered objects.
- **Extend Face Support:** Modify the program to handle quad faces and other polygons by triangulating them during loading.
- **Improve Clipping:** Implement proper clipping algorithms (e.g., Cohen-Sutherland or Sutherland-Hodgman) to handle vertices outside the viewport.
- **Enhance Camera and Object Manipulation:** Add advanced camera controls (e.g., panning, orbiting) and support for fine-grained editing of individual vertices or faces.

Bibliography

[1] Joey de Vries, *Learn OpenGL*. Retrieved from https://learnopengl.com.

[2] Mauricio Poppe, *Projection Transformations in Computer Graphics*. Retrieved from https://www.mauriciopoppe.com/notes/computer-graphics/viewing/projection-transform/.

[3] Donald Hearn, M. Pauline Baker, *Computer Graphics: C Version*. Prentice Hall, 1997.

[4] OpenAI, *ChatGPT*. AI Language Model, OpenAI, 2025.