3D Graphics Engine Theory and Design Process

Department of Mechanical and Mechatronics Engineering

A Report Prepared For:
The University of Waterloo

Prepared By:
Rakin Mohammed

January 23, 2022

Prof. William Melek
Director of Mechatronics Engineering, Undergraduate Studies
Department of Mechanical and Mechatronics Engineering University of Waterloo
200 University Ave W.
Waterloo, Ontario, N2L 3G1

Dear Professor Melek,

This report, entitled “3D Graphics Engine Theory and Design Process”, was prepared to fulfill my 3A work term report requirements. This is my 2nd work term report submission.

The intent of this research report is to provide you with context on the implementation and design choices made to program a 3D Graphics Engine in C++. This was a personal project I worked on during my co-op term, and this report will cover the decision process for choosing the programming language, an introductory analysis on 3D/2D graphics theory, as well as the implementation and optimization of line drawing algorithms. This report will also detail linear algebra and mathematical concepts required for matrix transformations and computer graphics representation.

This report was written entirely by me and has not received any previous academic credit at this or any other institution. I would like to acknowledge the help of Erfan Huq and Ishan Ghimire, 3A students at the University of Waterloo, who both helped me write code for the C++ graphics engine. They also provided explanations for implementation methods used for this project whenever questions arose. My role in the project was to research 3D/2D graphics theory and implement matrix transformations and line drawing algorithms.

Best Regards

Rakin Mohammed
3A Mechatronics Engineering

Table of Contents

List of Figures ii
List of Tables iii
Summary iv
1.0 Introduction 1
1.1 Project Requirements and Goals 1
1.2 Selecting Programming Language 1
1.3 Objective 2
2.0 Fundamental Concepts in Context of Computer Graphics 2
2.1 Euclidian Distance 2
2.2 Vectors, Normalization, Dot Product and Cross Product 3
2.3 Linear Interpolation and Simple Motion 5
2.4 Representation of Lines and Planes 6
3.0 Design Decisions 6
3.1 Architecture 6
3.2 Viewing Frustum and Projection onto 2D 8
3.3 Back-face Culling 9
3.4 Illumination 9
3.5 Camera Control 10
3.6 Clipping Algorithm 12
4.0 Conclusions 13
4.1 Conclusions 13
4.2 Next Steps / Recommendations 13
References 15

List of Figures

Figure 1: Pythagoras Theorem applied to find distance between two points. 3
Figure 2: System Architecture. 7
Figure 3: View Frustrum. 8

List of Tables

Table 1: Programming Language Comparison. 2
Table 2: Clipping Categorization Table.....12

Summary

To implement a 3D graphics engine from scratch, the programming language C++ was chosen due to its incredibly fast run-time speeds. Compilation speed is extremely important for triangular tessellation of polygons, as lots of polygons must be drawn and shaded. We can use the properties of vectors, matrices, and linear algebra to transform objects and determine the similarity between vectors. The similarity of vectors, or the dot product, is used for illumination and back-face culling, which is the process of determining which objects are not represented in the view space. A screen displays a 2D rectangle with a specific aspect ratio, representing a 2D projection of 3D images, which is called view space. However, the human eyesight interprets a 3D quadrilateral which is called the view frustrum or world space. A transformation matrix is derived to transform the camera from the origin to a point in 3D space, and point at a specific object, without implementing rotation. The inverse of this transformation matrix allows us to convert 3D information of points in world space to 2D information in view space.

1.0 Introduction

The video game industry has rapidly expanded ever since Willian Higinbotham developed “Tennis for Two” in 1958, dating back to be one of the first video games [1]. His goal was to show society that technological advances in computer graphics could bring positive value and “lighten up the place.” With the industry bringing in sales of $60.4billion in 2021 and with a market of over 227 million people in the US alone [2], many aspiring kids look to pursue a programming career to develop video games of their own. As a similarly inspired University of Waterloo student, I looked to learn 2D/3D graphics theory and implement my own 3D graphics engine to understand how game engines APIs like OpenGL and computer graphic technologies like ray tracing work.

1.1 Project Requirements and Goals

The goal of the project was to create a useable graphics engine API from scratch to efficiently render 3D data onto a 2D screen in the console of the chosen platform. The first problem to tackle is arranging a dataset of vertices into triangles, to represent an object graphically. This is called tessellation. Triangles are used as they are the simplest 2D shape, meaning you can create any polygon with just triangles. Next, each triangle in the 3D arrangement must be projected onto a flat plane in the perspective of the viewer. After projecting the image, lighting, back-face culling, clipping planes and controlling camera perspective must be implemented.

1.2 Selecting Programming Language

The three relevant programming languages for implementing computer graphics are C++, Python and Java. The biggest factor is the compilation speed of the language as lots of code must be processed to display graphics. This gives an advantage to low-level compiled languages, over high-level interpreted languages. The following decision matrix was constructed to decide as shown in Table 2.

Table 2: Programming Language Comparison [3]

Factors	Compiled Language	Compilation Speed	Level of Abstraction	Supports Operator Overloading	Availability of 3D APIs	Cross Platform Functionality	Total
Total Weight	**1**	**8**	**4**	**1**	**4**	**5**	**23**
C++	1	8	4	1	4	1	19
Python	0	2	3	1	1	3	10
Java	1	6	1	0	2	5	15

C++ was selected as the programming language of choice due to its incredibly fast compilation speed, and extensive libraries for outputting simple graphics/colours and 3D support. C++ is also notable for its efficient memory usage which will be useful when loading large numbers of triangles [3].

1.3 Objective

The objective of this project is to create an API that can use the input of an OBJ file to project 3D graphics on a 2D screen. The API must also allow users to view the object from different angles using keyboard input, maintaining correct shading.

The project must implement world transformation, that allows 3D objects to move around in the scene, view transformation, that allows the camera to be moved and positioned in the scene, and projection transformation which allows for the projection of 3D images onto a 2D plane.

2.0 Fundamental Concepts in Context of Computer Graphics

2.1 Euclidian Distance

The distance between two points in Euclidian space can be derived using the Pythagoras' Theorem, using the horizontal and vertical length of a triangle to find its hypotenuse. This concept can be used in 2 dimensions up to n dimensions trivially by repeatedly applying the Pythagoras' Theorem. This allows us to draw and find the magnitude of lines in 2D space. The 2D example is shown in Figure 1.

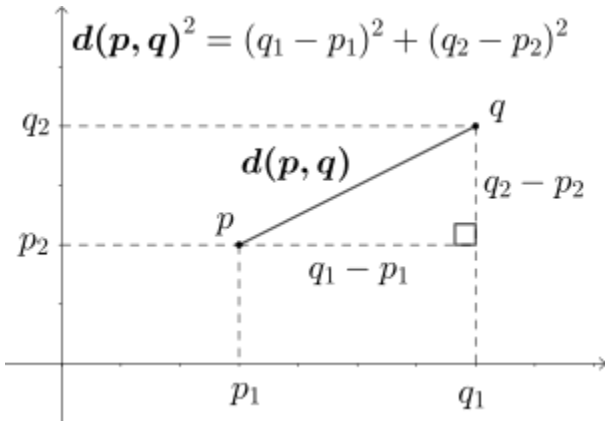

Figure 1: Pythagoras Theorem applied to find distance between two points [4]

The equation for distance between two cartesian coordinates in two dimensions is shown in Eqn. (1).

$$d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2} \quad (1)$$

The equation for distance between two points in three dimensions is shown in Eqn. (2).

$$d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + (q_3 - p_3)^2} \quad (2)$$

2.2 Vectors, Normalization, Dot Product and Cross Product

Vectors become very useful when simulating moving an object from position to another as it gives a magnitude as well as a direction. Assuming the tail of the vector is at the origin, given vector $\overrightarrow{V_1} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$, we can express its magnitude as shown in Eqn. (3).

$$\left|\overrightarrow{V_1}\right| = \sqrt{{x_1}^2 + {y_1}^2} \quad (3)$$

If multiple vectors are represented to move an object, we can take the sum of the magnitudes of the vectors to find the total distance travelled by the object, as shown in Eqn. (4).

$$Total\ Distance = \left|\overrightarrow{V_1}\right| + \left|\overrightarrow{V_2}\right| + \cdots + \left|\overrightarrow{V_n}\right| = \sqrt{{x_1}^2 + {y_1}^2} + \sqrt{{x_2}^2 + {y_2}^2} + \cdots + \sqrt{{x_n}^2 + {y_n}^2} \quad (4)$$

To determine the final position vector of our object, $\vec{P} = \begin{bmatrix} P_x \\ P_y \end{bmatrix}$, we can perform simple vector addition as shown in Eqn. (5).

$$Final\ Position\ Vector = \vec{P} = \overrightarrow{V_1} + \overrightarrow{V_2} + \cdots + \overrightarrow{V_n} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} + \cdots + \begin{bmatrix} x_n \\ y_n \end{bmatrix} =$$

$$\begin{bmatrix} P_x \\ P_y \end{bmatrix} = \begin{bmatrix} x_1 + x_2 + \cdots + x_n \\ y_1 + y_2 + \cdots + y_n \end{bmatrix} \quad (5)$$

Unit vectors are essential for determining the path an object will follow, as normalizing a vector provides the direction vector of an object. Unit vectors also allow for scaling of objects and movement vectors. The general expression for a unit vector is shown in Eqn. (6).

$$\hat{V} = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} \\ \frac{y}{\sqrt{x^2+y^2}} \end{bmatrix} \quad (6)$$

Using a unit circle and trigonometry, we can also represent a unit vector with the angle it makes with the positive x-axis. This is known as polar co-ordinates and is shown in Eqn. (7).

$$\hat{V} = \begin{bmatrix} \cos(\emptyset) \\ \sin(\emptyset) \end{bmatrix} \quad (7)$$

A point on the direction vector, p, can be represented by multiplying the unit vector $\hat{V}$ by the distance away it is from the origin. When rotating objects, we will work with radians and polar co-ordinates, whereas we'll use vectors for translating objects.

The dot product of two vectors is defined as the sum of the product of the vector components. Dot product will only be used with unit vectors in this report to use its specific properties. This formula outputs a scalar value, which is used to determine how much of a vector project onto another. The dot product of $\widehat{V_1}$ and $\widehat{V_2}$ is expressed in Eqn. (8).

$$Dot\ Product = \widehat{V_1} \cdot \widehat{V_2} = V_{1x} * V_{2x} + V_{1y} * V_{2y} \quad (8)$$

The dot product of two unit vectors is very useful in programming graphics. For example, we can calculate the angle between two vectors by using the dot product as shown in Eqn. (9).

$$\emptyset = \cos^{-1}(\widehat{V_1}.\widehat{V_2}) \quad (9)$$

Dot product can also be used to determine the similarity in direction of two vectors. If you take the dot product of a unit vector with itself, it will return a scalar value of 1. If you take the dot product of a unit vector with its normal, it will return a scalar value of 0. If you take the dot product of a unit vector with its opposite vector, it will return a scalar value of -1. This is extremely useful for movement in graphics, as if you define a desired path vector, $\hat{D}$, you can determine an approximate direction the true path vector, $\hat{T}$, is moving relative to $\hat{D}$, using the dot product. An example is if the dot product of the desired path and true path tends to 1, it is moving in a similar direction.

To generate the normal vector of two vectors or a triangular surface, we take the cross product of the vectors as shown in Eqn. (10) [5].

$$\vec{A} \times \vec{B} = [(A_y * B_z - A_z * B_y),\ (A_z * B_x - A_x * B_z),\ (A_x * B_y - A_y * B_x] \tag{10}$$

2.3 Linear Interpolation and Simple Motion

Given two points (x_1, y_1) and (x_2, y_2), linear interpolation can be used to find any point between x_1 and x_2. In the context of an object moving through time, we can introduce a parameter $t, 1 \geq t \geq 0$. To find a point on the line, P_t, we can represent its components with the following expressions as shown Eqn. (11) [6].

$$P_{tx} = x_1 + (x_2 - x_1) * t$$
$$P_{ty} = y_1 + (y_2 - y_1) * t \tag{11}$$

It is possible to define our timesteps, or frames, using the parameter t. If we let the time elapsed per frame to be T_{pf}, we can introduce a counter for elapsed time such that $T_e = T_e + T_{pf}$. We can then set our parameter using the total runtime, T_t, where $t = \frac{T_e}{T_t}$, fulfilling our parametrization.

Linear interpolation allows us to intuitively implement controlled movement, allowing us to move objects in a set path. Another important use of linear interpolation is when working with difficult to derive functions that one might have to work with. An example of this would be complex sound waves. Linear interpolation allows us to approximate data points on the graph and smooth out the curve. Note that we can also further extrapolate the line between the two points to find where our object will be on the path in the future.

Introductory kinematics are used to implement simple motion in the graphics engine. Given an objects current position, P_t, its velocity vector, V_t, and its acceleration vector, $A_t = \begin{bmatrix} 0 \\ -9 \end{bmatrix}$ (Approximation of gravity), we can represent the movement of an object with the following expressions in Eqn. (12) and Eqn. (13).

$$P_{t+1} = P_t + V_t * t \tag{12}$$

$$V_{t+1} = V_t + A_t * t \tag{13}$$

2.4 Representation of Lines and Planes

Let a set of solutions be represented by the line, $y = mx + c$. Substituting $m = \frac{dy}{dx}$, we get the following expression, $-dyx + ydx = c$. The perpendicular gradient to the line is given by the negative inverse of m. Let the perpendicular be represented by $\vec{N}$. It follows then, $\frac{-dx}{dy} = \frac{-N_x}{N_y}$. Using the gradient of the normal in our initial expression, we get $xN_x + yN_y = C$.

Given a vector to a point on the line, $\vec{P}$, and the normal, $\vec{N}$, we can find determine the 2D line of solutions including point P using the following expression, $xN_x + yN_y - \vec{P} \cdot \vec{N} = 0$. This can be easily scaled for 3D planes by introducing zN_z as a summand to the expression.

3.0 Design Decisions

3.1 System Architecture

This project was designed and implemented using GitHub as a version control tool. The file architecture of the project is shown in Figure 2.

Figure 2: System Architecture

The first class to implement is the “gameEngine.h” class, which is used to set up the console window for graphics, interpret and display error messages, allow user keyboard input, and manipulate functions from other classes. After that, there are five levels of abstraction used to organize the architecture of the C++ code.

The first three classes deal with the math required for transformations of vectors and matrices. The first class “Math.h” abstracts the implementation of vector arithmetic including dot product, normalization of vectors, and vector plane intersections. The second class, “Transformation.h” abstracts the implementation of rotation and translation algorithms of a four by four matrix. Lastly, the class “matrixMath.h”, abstracts the implementation of matrix multiplication and vector matrix multiplication.

The two other notable classes deal with drawing and manipulating polygons. The class “Tri.h” is responsible for arranging and transforming triangles from groups of vertices. It is also responsible for the implementation of clipping and determines whether parts of a triangle would be inside or outside the viewing frustum. The last class, “Object.h”, does most of the heavy lifting when it comes to representing graphical information. This class implements the functions required to draw lines, triangles and fill in polygons. It is also responsible for the implementation of view in perspective, illumination, as well as the DDA algorithm required to rasterize lines, triangles, and polygons.

3.2 Viewing Frustum and Projection onto 2D

The screen a viewer sees will be a rectangle with a specific width and height. To normalize the screen space, we can let the origin be (0, 0) in the center of the screen. The left side of the screen will have a midpoint of (-1, 0) and the top side of the screen will have a midpoint of (0, -1). With this normalized screen space, we can determine that objects above +1 and below -1, will not be represented. The screen space will be referred to as view space. We can define an aspect ratio $a = \frac{h}{w}$, to scale movements within the x screen space accordingly.

However, the perspective of the camera will be in the shape of a trapezoid to accurately represent human vision. This is also known as a viewing frustrum or field view as shown in Figure 3. The viewing frustrum will be referred to as world space.

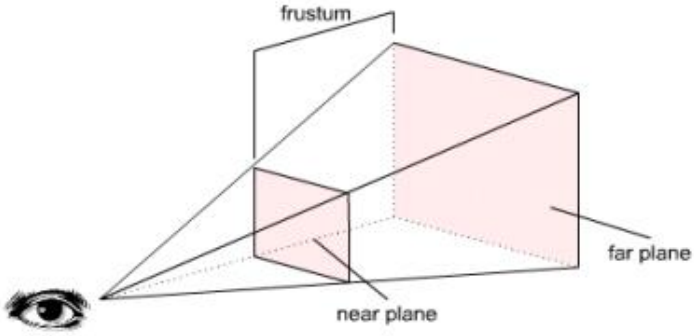

Figure 3: View Frustrum [7]

The far clipping plane is restricted from [-1, +1] just the same as the near clipping plane. However, since the far clipping plane is viewed as further away, the object in sight must be scaled smaller than closer objects and vice versa. We can represent the angle of view as ∅. It is intuitive that as ∅ increases, the width of the far plane increases. We can represent the relationship between the opposite and adjacent sides of the viewing angle with the following expression, $F = \frac{1}{\tan\left(\frac{\emptyset}{2}\right)}$ to get our intended scaling factor for x and y.

The methods to normalize z is simple. Let Z_{far} represent the distance from the far plane to the viewer and let Z_{near} represent the distance from the near plane to the viewer. To scale objects in the z plane, we normalize with the following expression, $z * \frac{Z_{far}}{Z_{far} - Z_{near}} - \frac{Z_{far} * Z_{near}}{Z_{far} - Z_{near}}$. The last scaling factors required is scaling x and y relative so z, or in other words, how far away the object is from the screen. This is done by dividing x and y by z accordingly.

We can represent our 3D vector as $[x, y, z]$. Our 2D vector are represented using our aspect ratio and scaling factors as shown in Eqn. (14).

$$Let\ q = \frac{Z_{far}}{Z_{far} - Z_{near}}$$

$$[x, y, z] \rightarrow [\frac{a*F*x}{z},\ \frac{F*y}{z},\ z*q - Z_{near*}q] \tag{14}$$

To go from our 3D vector to 2D vector, we can model a transformation matrix based on Eqn. (14). To represent a fourth element in the matrix, given three variables $[x, y, z]$, we can introduce a 1, giving a 3D vector of $[x, y, z, 1]$. This 3D vector is transformed using the following matrix as shown in Eqn. (15)[1].

$$\begin{bmatrix} aF & 0 \\ 0 & F \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$
$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad \begin{bmatrix} q & 1 \\ -Z_{near}q & 0 \end{bmatrix} \tag{15}$$

3.3 Back-face Culling

In computer graphics, back-face culling is the process of determining whether a polygon is visible to the viewer [8]. Using the cross product formula outlined earlier in the report, we can find the normal of each triangular face and inspect its z-component relative to the line of projection from the camera to the location of the normal. The angle created with the line of sight to the object and the normal, tells us which polygons we can't see if the angle created is greater than or equal to 90°. As described earlier, we can take the dot product of the normal and the line of sight to compare how similar the two vectors are. If the dot product of the surface normal and the camera to triangle vector are greater than or equal to 0, the viewer cannot see the object and the triangle is discarded.

3.4 Illumination

In computer graphics, real-time lighting is categorized into static and dynamic shading. Static shading is when the shading effect permanently colours the 3D object and does not change during run-time and is very useful when designing world models. For this project, I chose to implement dynamic shading, as it is the method used to continuously update the shading of moving 3D objects. This process uses simple vector operations to modulate the surface color of a polygon based on its angle to the light direction [7]. We can use the dot

[1] The inner brackets of the 4 by 4 matrix are redundant and are used for clarity

product of the unit vector facing towards the and the normal of a triangular surface to determine the similarity of the two vectors. If the vectors are similarly aligned, and the dot product is between 0 and 1, then the surface will be illuminated. The surface color is scaled by the intensity of the illumination from 0 to 1. In simple terms, if the perpendicular of a surface is facing the same direction as where the light is coming from, then it will be illuminated. If the perpendicular of a surface is facing away from the direction the light is coming from, then it will not be illuminated. This process is repeated for every triangle in a polygon to properly shade and color each surface. This method is called Lambert shading, which uniformly colours a polygon surface.

3.5 Camera Control

The two ways to implement a moving camera is by either moving the camera around the scene or alternatively, moving the entire scene around the camera. The former is chosen for this project, as the center of rotation of objects in the scene are not static, making it difficult to rotate every object in a scene. To rotate the camera in space, we look to derive a transformation that translates the head of the line of sight vector towards the point to look at, about the tail of the vector, called the eye point. Let $\vec{A}$ represent the forward vector of the camera, $\vec{B}$ represent the perpendicular to the forward vector, and $\emptyset$ represent the angle between $\vec{A}$ and the x-axis. As trigonometric functions are intensive and to save computational time, instead of applying a rotation transformation by $\emptyset$, we look to find a translation matrix required to move point P in cartesian co-ordinate system to its final position Q. To do this, we take the forward vector components of $\vec{A}$ and $\vec{B}$ represent the x and y-axis of its own cartesian system. Point Q has the same co-ordinates as P in the new space relative to $\vec{A}$, however, the co-ordinates of Q, are different from P in the original cartesian system. The x-component of Q in the normal cartesian system is defined as the x-component of $\vec{A}$ scaled by the point P plus the x-component of $\vec{B}$ scaled by the point P. This is the same case for the y-component and the transformation for points outlined in Eqn. (16).

$$Q = \begin{bmatrix} Q_x \\ Q_y \end{bmatrix} = \begin{bmatrix} P_x A_x + P_y B_x \\ P_x A_y + P_y B_y \end{bmatrix}$$
$$= \begin{bmatrix} P_x \, P_y \end{bmatrix} \begin{bmatrix} A_x & A_y \\ B_x & B_y \end{bmatrix} \quad (16)$$

To represent this translational rotation in 2D, we introduce another dimension to our matrix. Letting $\vec{T}$ represent the translation offset of the eye point from the origin, the transformation matrix to determine point Q derived from rotating point P by $\emptyset$ is shown in Eqn. (17).

$$Q = \begin{bmatrix} P_x & P_y & 1 \end{bmatrix} \begin{bmatrix} A_x & A_y & 0 \\ B_x & B_y & 0 \\ T_x & T_y & 1 \end{bmatrix} \tag{17}$$

To represent this translational rotation in 3D, we introduce yet another dimension to our matrix. Letting $\vec{C}$ represent the orthogonal vector to both $\vec{A}$ and $\vec{B}$, the transformation matrix used to determine point Q is shown in Eqn. (18)[2].

$$Q = \begin{bmatrix} P_x & P_y & P_z & 1 \end{bmatrix} \begin{bmatrix} A_x & A_y \\ B_x & B_y \end{bmatrix} \begin{bmatrix} A_z & 0 \\ B_z & 0 \end{bmatrix} \begin{bmatrix} C_x & C_y \\ T_x & T_y \end{bmatrix} \begin{bmatrix} C_z & 0 \\ T_z & 1 \end{bmatrix} \tag{18}$$

The transformation matrix outlined in Eqn. (18) will be referred to as the "Point at Matrix". This function is implemented to transform the line of sight vector to point the vector towards the point it should look at, Q, as well as translate it from the origin. Now that we have implemented the function to move and control the camera vector in 3D world space, it is necessary to do the inverse of this transformation matrix to return all points to 2D view space. This means transforming all points back to their original position relative to the camera looking from the origin, as it was prior applying the translation and rotations on the camera vector. Since the "Point at Matrix" uses translation and rotation, and not scaling, we are able to take its inverse giving the following translation matrix outlined in Eqn. (19) [9].

$$Q = \begin{bmatrix} P_x & P_y & P_z & 1 \end{bmatrix} \begin{bmatrix} A_x & B_x \\ A_y & B_y \end{bmatrix} \begin{bmatrix} C_x & 0 \\ C_y & 0 \end{bmatrix} \begin{bmatrix} A_z & B_z \\ -\vec{T} \cdot \vec{A} & -\vec{T} \cdot \vec{B} \end{bmatrix} \begin{bmatrix} C_z & 0 \\ -\vec{T} \cdot \vec{C} & 1 \end{bmatrix} \tag{19}$$

The transformation matrix outlined in Eqn. (19) will be referred to as the "Look at Matrix". This function is implemented to invert the "Point at Matrix", allowing us to transform any object co-ordinates in world space to into view space from the perspective of the camera at the origin.

[2] The inner brackets of the 4 by 4 matrix are redundant and are used for clarity

3.6 Clipping Algorithm

Let's assume we have a line where every triangle on the far side of the line, we want to display, and everything on the near side, we want to discard. There are three scenarios in which the triangle can be arranged.

1. The triangle can be entirely on the far side of the line. These triangles are displayed.
2. The triangle can be entirely on the near side of the line. These triangles are not displayed.
3. The triangle can intersect with the line. Part of the triangle is displayed.
    a. The resulting polygon forms a triangle with the line
    b. The resulting polygon forms a quadrangle (an unenclosed quadrilateral). Split the quad into two triangles from opposing vertices

The algorithmic process of doing this is to count the number of points on the far side of the line, not including points on the line. This allows us to determine which category of arrangement the triangle falls under as shown in Table 3.

Table 3: Clipping Categorization Table

#Of Points on the Inside of Line	**0**	**1**	**2**	**3**
Decision	Reject Triangle	Form Triangle	Form Quad	Accept Triangle
New Triangle Formation	N/A	Create triangle from point inside the line, and the two intersection points on the line	Create triangle from two points inside the line, and one of the intersection points on the line. Create another triangle using one point inside the line and two points on the line.	N/A

If we check for clipping of triangle on our clipping plane, it is required to check every edge of the plane for clipping. In the world view example, we will test against the near clipping plane and the far clipping plane. It is also noted that each instance of clipping will create additional triangles. To implement this, we create a function that returns the requisite triangles to satisfy a clip, as well adds the triangles to a queue for processing. The pseudocode for implementing this algorithm is briefly described.

1. Generate a queue of triangles to be clipped
2. For each plane to be clipped against, do the following:
3. For each triangle in the queue, do the following:
4. Remove the triangle at the front of the queue
5. Test the triangle for clipping against the plane, generating 0 to 2 new triangles
6. Navigate to next triangle
7. Generate a queue for new triangles created by clipping, testing against plane not required
8. Navigate to next plane

4.0 Conclusions

4.1 Conclusions

Using fundamental 2D/3D graphics theory and Linear Algebra, we were able to implement a C++ graphics engine that can read OBJ files, and fully render and move 3D objects in 2D. The API allows for user controlled camera movement, while accurately modulating polygon illumination. Although there are different design choices in this project, such as using a different programming language with unique graphic libraries, or using different line traversal methods, the engineering design process was followed resulting in me achieving the objective of displaying 3D graphics and controlling a camera view in a C++ console.

4.2 Next Steps / Recommendations

For future study, it is recommended to create a criterion for render run-time. As compilation efficiency is the most important thing for a graphics software to be useable and marketable, analyzing the run-time speeds of different versions of the project can provide a benchmark used to test the optimization of the game engine. The optimization of the engine must undergo continuous validation to ensure that the code isn't wasting memory and computing power. Avoiding using trigonometry in C++ is one way to reduce run-time, as matrix and vector transformations are much more efficient.

Another area of improvement for the graphics engine is to use Gouraud shading instead of Lambert shading. This technique allows for smooth and continuous shading using interpolation of the illumination across a polygon. This is done by determining the average unit normal vector of adjoined faces, at each vertex of a polygon. Then, the vertex intensity is calculated using the dot product of the new normal. Lastly, linear interpolation is performed across the 3D mesh and is used to determine the intensification across a polygon face using a scan line [10]. Utilizing Gouraud shading will allow for objects to be represented much smoother in the graphics engine.

References

[1] Brookhaven National Laboratory, "Brookhaven National Laboratory," [Online]. Available: https://www.bnl.gov/about/history/firstvideo.php#:~:text=Before%20'Pong%2C'%20There%20Was%20'Tennis%20for%20Two'&text=Tennis%20for%20Two%20was%20first,an%20oscilloscope%20for%20a%20screen.. [Accessed 23 01 2022].

[2] Entertainment Software Association, "ESA," Entertainment Software Association, [Online]. Available: https://www.theesa.com/news/u-s-consumer-video-game-spending-totaled-60-4-billion-in-2021/. [Accessed 23 01 2022].

[3] Software Testing Help, [Online]. Available: https://www.softwaretestinghelp.com/python-vs-cpp/. [Accessed 23 01 2022].

[4] Wikipedia, "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Euclidean_distance. [Accessed 23 01 2022].

[5] MathIsFun, "Math Is Fun," [Online]. Available: https://www.mathsisfun.com/algebra/vectors-cross-product.html. [Accessed 23 01 2022].

[6] Wikipedia, "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Linear_interpolation. [Accessed 23 01 2022].

[7] Multimedia Applications Division Freescale Semiconductor, Inc, "3D Math Overview and 3D Graphics," 2010.

[8] Wikipedia, "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Back-face_culling. [Accessed 23 01 2022].

[9] CUEMATH, "CueMath," [Online]. Available: https://www.cuemath.com/algebra/inverse-of-a-matrix/. [Accessed 23 01 2022].

[10] Java Point, "JavaPoint," [Online]. Available: https://www.javatpoint.com/computer-graphics-gouraud-shading. [Accessed 23 01 2022].

[J. Academia, "Lecturing Technical Writing to 1st Year Engineering Students," *Journal of Engineering*
1 *Education,* p. 6, 2017.
1
]

[S. Magoo, "An Analysis of the Impact of Teaching Methods on Student Academic Performance," in
1 *Engineering Education Conference*, Seattle, 2015.
2
]

[IEEE, "IEEE Referencing: A Users Guide," Waterloo Publishing, Waterloo, 2017.
1
3
]