# CPU Graphics Engine

Thomas Dizon

November 7, 2025

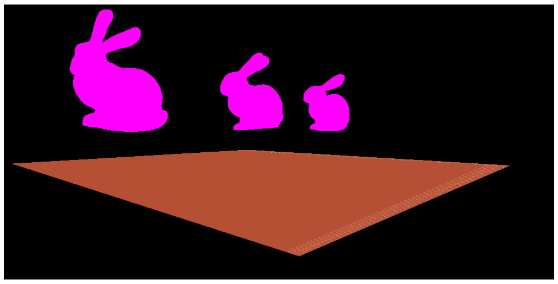

Figure 1:

## Contents

# 1 Introduction

## 1.1 Motivation

- As someone interested in developing 3D games and graphics, I was frustrated with how much I didn't understand when I used game engines like Unity and Godot. When I wanted to achieve a simple effect like adding grass, water or natural-looking terrain, I was overwhelmed with my own inability. Thus, I committed myself to building a 3D engine from scratch so I could allow these concepts that I didn't understand emerge naturally as byproducts as I worked towards the goal of rendering 3D objects.
- Now that I've more or less finished the project, it is A LOT easier now to imagine how to achieve some graphical effect like a grass or water shader. Since I understand how it works conceptually under the hood, the programming is a lot easier.
- It's also a great precursor to understanding more in depth how the GPU works.

## 1.2 What is a *Graphics Engine*?

- The 'rasterization' step in the graphics render pipline is a specific part which involves converting Primitives into Fragments
- A primitive, in my case, is a Triangle and a Fragment is an (x,y) value on the screen with an additional z value to track its depth
- In the real world, rasterization is done purely through specialized hardware on the GPU (which is why it is so fast)
- To conceptually understand how it works, instead of building a GPU from scratch (hard), I'll use my good old CPU to do the rasterizing (still hard but easier)

## 1.3 Why *C*?

- I first touched C during my Systems Programming course at University. It was extrememly difficult for me at the time but very benefiical for my programming skills. I felt that I haven't extracted all that I can from the language yet.
- Also, I wanted to get as low level as possible for this project to ensure I had minimal dependencies. To truly understand the Graphics Pipeline, I wanted to build it truly (well, mostly) from scratch

### 1.4 What does the project depend on?

- Right now, it depends on the C standard library, std_image.h (reference) for dealing with .png files and SDL2 (reference) for handling all the OS level stuff like inputs and writing to the screen. Though you could easily replace SDL with any other library which creates Windows for you like RayLib.

## 2 The Graphics Pipeline

### 2.1 Diagram

### 2.2 Overview

- Primitive Assembly
- Vertex Shader
- Clipping & Viewport
- Rasterization
- Fragment Shader
- Output Merging & Depth Test

## 3 3D Geometry in Memory

### 3.1 How do you represent a 3D object on a computer?

#### 3.1.1 Vertices and Triangles

- Using a Mesh

#### 3.1.2 3D Mesh

### 3.2 3D File types

## 4 Coordinate Spaces

### 4.1 What is a *Coordinate Space*?

### 4.2 GameObjects & Object Space

### 4.3 Scenes & World Space

### 4.4 Cameras & View Space

## 5 Transforming Vertices

### 5.1 Why do we need 4 dimensions?

### 5.2 Matrices as Transformations

### 5.3 Quaternions as Rotations

### 5.4 The Model Matrix

### 5.5 The View Matrix

### 5.6 The Projection Matrix

### 5.7 The Viewport Matrix

### 5.8 The Vertex Shader

### 5.9 Clipping & The Sutherland Hodgman Algorithm

## 6 Drawing to the screen

### 6.1 What is *Rasterization*?

### 6.2 Interpolation

### 6.3 Barycentric Coordinates

### 6.4 Rasterizing Triangles

### 6.5 What is a *Fragment?*

### 6.6 The Fragment Shader

## 7 Shading

### 7.1 Materials

- A Material

- It contains ...

### 7.2 Textures

### 7.3 Unlit Shader

### 7.4 Lit Shader

### 7.5 Phong Shader

### 7.6 Toon Shader

### 7.7 Shadows

### 7.8 Ambient Occlusion

## 8 Deprecated

### 8.1 Math Concepts

#### 8.1.1 Vectors

- In this document, we will use typical Vector operations and notation including but not limited to the following
- A vector is **v** is a collection of real numbers denoted by
- The *dot product* is defined by
- The *cross product* is defined by
- A unit vector is denoted by the hat

#### 8.1.2 Planes

- A plane is defined by ****TODO****

#### 8.1.3 Matrices

- To understand Graphics Engines, you must become somewhat familiar with the operation of matrices. In particular: Matrix Multiplication, Inverse Matrices, Matrices as Linear Transformations and probably more. I provide a very scarce review below.
- Matrix multiplication is done like so:
- Matrix Inverse is calculated like so:
- For an affine, orthonormal matrix, the inverse can be calculated like this ( + reference)

### 8.1.4 Quaternions

- To represents rotations in 3D space, I use Quaternions. While you can use a composition of 3x3 Euler Matrices, they are slower, hard to interpolate and are at risk of Gimbal Lock (+ Reference).
- A Quaternion is a kind of 4D complex number $q = q_0 + q$ where $q_0$ is a scalar and $q$ is an R3 vector where each component has unit vectors $i, j, k$.
- The unit vectors $i, j, k$ adhere to the following multiplication rules. ****TODO****
- Whenever you are interested in rotating an object in 3D space, you need to specify 2 things: An axis and an angle. These are encoded in the Unit Quaternion in the following way: ****TODO****
- In the space of Unit Quaternions, if you operate a Unit Quaternion on a Vector, it results in a rotation of some degrees about some axis.

### 8.1.5 Coordinate Spaces

- Cartesian
- Projective
- Barycentric
- Spherical, Cylindrical

### 8.1.6 Transformations

- Model, View, Projection

### 8.1.7 Deriving the Model Matrix

### 8.1.8 Deriving the View Matrix

### 8.1.9 Deriving the Projection Matrix

The Projection Matrix is responsible for converting vertices from View space into Clip space. It aims to morph points that lie inside or on the View frustum into the Canonical View Volume in such a way that *perspective* is introduced.

Perspective is described simply by the idea that things that are far away seem small and things that are close seem big.

We break the problem up into 2 parts:

1. Calculating the transformation for $x$ and $y$
2. Calculating the transformation for $z$

For the first part, consider the setup in Figure 2. In this 2D slice, the viewing volume is given by the near and far planes and the top and bottom planes. What we wish to do is to take some point $v$, and project it onto the near plane.

To do this, we draw a line from the point to the origin. Consider the point at which this origin ray intersects the near plane. Let's call this point $x$.

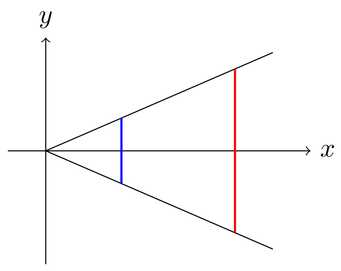

Figure 2: 2D slice depicting the top, bottom, near and far planes

#### 8.1.10 Deriving the Viewport Matrix

## 8.2 Algorithms

### 8.2.1 Rasterization

### 8.2.2 Improved Rasterization

- (First Approach) Takes a Primitive (Triangle made of Vertices) after is has been transformed by MVP and VP. It computes Bounding Box and using BaryCentric Coordinates with respect to the Triangle vertex positions decides which pixels to remove based on whether they are inside or outside of the triangle ****FIGURE****
- (Optimized Approach) Iterative Edge Functions ****TODO****

### 8.2.3 Clipping

- After the Vertex Shader is applied, triangle vertices should be in *Clip Space*, which is a *homogeneous vector space* in $R^4$ where $v = (x, y, z, w)$ maps to $u = (x/w, y/w, z/w)$ in $R^3$
- The goal of this algorithm is to *clip* the triangle against the *View Frustum*, which in homoegenous space is a volume bounded by 6 4D planes.
- We start by deriving an expression for each planes, represented by a normal vector $n \in \mathbb{R}^4$ and some point on the vector $p \in \mathbb{R}^4$
- The projection matrix maps each of the 6 $\mathbb{R}^3$ planes that define the view frustum. ****TODO**** Show that the 6 view frustum planes map to the planes used in 4 space

- ****TODO**** Write some pseudocode, include a diagram

## 9 Conclusion & Future Work

### 9.1 What I learnt

### 9.2 Limitations

### 9.3 A future implementation in C++

## 10 References