CPU Graphics Engine

Thomas Dizon

November 10, 2025

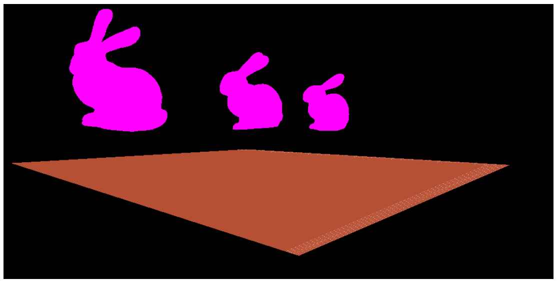


Figure 1:

Contents

1 Introduction 5

- 1.1 Motivation 5
- 1.2 What is a *Graphics Engine*? 5
- 1.3 Why *C*? 5
- 1.4 What does the project depend on? 6

2 The Graphics Engine 6

- 2.1 Diagram 6
- 2.2 Overview 6

3 3D Geometry in Memory 6

- 3.1 How do you represent a 3D object on a computer? 6
- 3.2 Vertices and Triangles 6
- 3.3 3D Mesh 7
- 3.4 3D File types 7

4 Coordinate Spaces 7

- 4.1 What is a *Coordinate Space*? 7
- 4.2 GameObjects & Object Space 7
- 4.3 Scenes & World Space 7
- 4.4 Cameras & View Space 7

5 Transforming Vertices 7

- 5.1 Matrices as Transformations 7
- 5.2 Deriving the Model Matrix 7
  - 5.2.1 Transforms 7
  - 5.2.2 Scale Matrix 7
  - 5.2.3 Rotation Matrix 7
  - 5.2.4 Translation Matrix 7
  - 5.2.5 Combining them together 7
- 5.3 Deriving the View Matrix 7
  - 5.3.1 The Matrix Inverse 7
  - 5.3.2 Inverse of the Model Matrix 7
- 5.4 Deriving the Projection Matrix 7
  - 5.4.1 What is *Perspective*? 8
  - 5.4.2 The View Frustum 8
  - 5.4.3 Perspective Projection 9
- 5.5 Deriving The Viewport Matrix 12
- 5.6 The Vertex Shader 12
- 5.7 Clipping & The Sutherland Hodgman Algorithm 12

6 Drawing to the screen 12
6.1 What is *Rasterization*? 12
6.2 Interpolation 12
6.3 Rasterizing Triangles 12
6.4 What is a *Fragment?* 12
6.5 The Fragment Shader 12
6.6 Depth Testing 12
6.7 The Framebuffer and Z-Buffer 12

7 Shading 12
7.1 Shaders 12
7.2 Materials 12
7.3 Textures 13
7.4 Unlit Shader 13
7.5 Lit Shader 13
7.6 Phong Shader 13
7.7 Toon Shader 13
7.8 Shadows 13
7.9 Ambient Occlusion 13

8 Appendix 13
8.1 Quaternions 13
8.2 Barycentric Coordinates 13

9 Deprecated 13
9.1 Math Concepts 13
 9.1.1 Vectors 13
 9.1.2 Planes 13
 9.1.3 Matrices 13
 9.1.4 Quaternions 14
 9.1.5 Coordinate Spaces 14
 9.1.6 Transformations 14
 9.1.7 Deriving the Model Matrix 14
 9.1.8 Deriving the View Matrix 14
 9.1.9 Deriving the Projection Matrix 14
 9.1.10 Deriving the Viewport Matrix 15
9.2 Algorithms 15
 9.2.1 Rasterization 15
 9.2.2 Improved Rasterization 15
 9.2.3 Clipping 15

10 Conclusion & Future Work 16
10.1 What I learnt 16
10.2 Limitations 16
10.3 A future implementation in C++ 16

11 References **16**

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)
- Just to clarify between similar terms. A "Software Rasterizer" refers to writing a CPU implementation of the Rasterization step in a "Software Renderer". A "Software Renderer" is a CPU implementaion of the Graphics Pipeline. A "Graphics Engine" is a CPU implementation of the graphics pipeline as well as Scene Management, Asset Management, Input handling, Updating the Scene etc. It falls short of being a fully fledged game engine but the scope is larger than a Software Renderer

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 Engine

2.1 Diagram

2.2 Overview

- Preparing Assets for the Scene
- Preparing the Scene
- Updating the Scene
- 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?

3.2 Vertices and Triangles

- Using a Mesh

3.3 3D Mesh

3.4 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 Matrices as Transformations

5.2 Deriving the Model Matrix

5.2.1 Transforms

5.2.2 Scale Matrix

5.2.3 Rotation Matrix

5.2.4 Translation Matrix

5.2.5 Combining them together

5.3 Deriving the View Matrix

5.3.1 The Matrix Inverse

5.3.2 Inverse of the Model Matrix

5.4 Deriving the Projection Matrix

The goal of this section is to come up with some matrix **P** that transforms vertices from View Space into Clip Space. We wish to do it in such a way that Perspective is captured, and that vertices inside the View Frustum are mapped to Normalized Device Coordinates.

Normalized Device Coordinates (NDC) is given by all points (x,y,z) such that $x \in [-1, 1]$, $y \in [-1, 1]$ and $z \in [0, 1]$ which matches the *Vulkan* convention.

Mathematically, we want to find P such that

$$\mathbf{Pv_{view}} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ z_{view} \end{pmatrix}$$

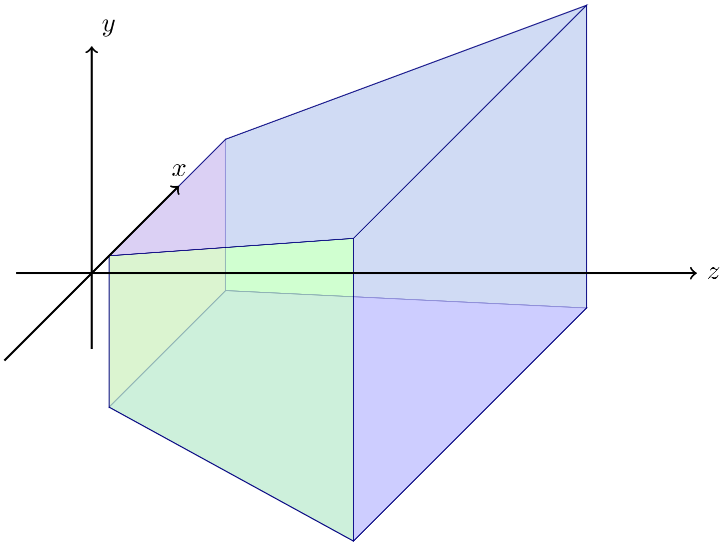
5.4.1 What is *Perspective*?

- In this project, my goal is to create graphics that are visually realistic or at least somewhat consistent with our experiences in the real world. Thus, my definition of Perspective will be made with that in mind.
- Put simply, Perspective is the phenomenon of objects appearing smaller to a viewer as it moves further away from the viewer.
- The viewer perceives the world through projecting a 3D world onto a 2D 'viewing plane'
- It comes from that fact that we, as humans, perceive a 3D world through light striking a 2D surface, our retina after passing through a 'pinhole' which is our iris.
- Our eye accepts light from all objects in the surrounding area in a 'radial' fashion. As a result, distant objects appear smaller, while nearby objects appear larger.
- Mathematically, the apparent size of an object is inversely proportional to its distance from the observer.

5.4.2 The View Frustum

- Since we are trying to create realistic graphics on a computer screen, we need to take that screen into account.
- We imagine that the computer screen is not a screen but instead a 'Window' into the virtual world we are creating. We can now place this Window into the Scene, in world space. The Width and Height of the window is proportional to the actual width and height of the Window running the program (Viewport)
- The distance between the viewer and the screen in World Space is given by the Field Of View paremeter which is an angular measurement.
- So now the image is: We, the viewer, are the Camera in the Scene and the Computer Screen is some distance directly in front of us, almost as we are in real life.
- Now, in world space, we can extend the Computer Screen out into infinite and call it the 'near plane'
- So that we don't blow up our computers by trying to render infinitely into the distance, we also define a 'Far Plane' based on the maximum distance from the Viewer we wish to see.
- Now, we can't see above, below, right or left of the window so we can trace a line from the Camera to each edge of the viewport and create The Top, Left, Right and Bottom planes from that.

- The objects inside of this volume are the objects visible to our Viewer.
- All vertices inside the view frustum will be mapped to points inside our Canonical View Volume



5.4.3 Perspective Projection

After applying the Model and View Matrix to each Vertex in the Scene, each Vertex should now be in View space (Coordinates expressed relative to the Camera being at the origin). We want to project each Vertex onto the Viewing Plane which is the same thing as the Near Plane. For now, we ignore the z and w values.

For each Vertex, imagine constructing a line from the Vertex to the Origin. This line's intersection on the Viewing plane will be the Vertex's projection.

The parametric vector representing a line passing through points A and B is given by

$$\mathbf{r}(t) = \mathbf{A} + t(\mathbf{B} - \mathbf{A}), t \in \mathbb{R}$$

In our case, $A = (0, 0, 0)$ and $B = v$. Thus,

$$\mathbf{r}(t) = \mathbf{v}t$$

To calculate where it intersects the near plane, we solve the equation for t after setting $r_z = n$, where n how far the near plane is along the z axis.

$$r_z = n \Rightarrow v_z t_{int.} = n \Rightarrow t_{int.} = \frac{n}{v_z}$$

Then,

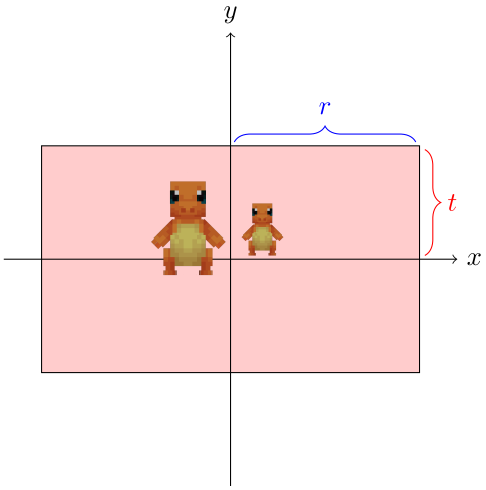
$$\mathbf{r}(t_{int.}) = \mathbf{v}(\frac{n}{v_z}) = (\frac{nv_x}{v_z}, \frac{nv_y}{v_z}, n)$$

In other words, the projected x and y values of the vertex $\mathbf{v}$ on the near plane will be nv_x/v_z and nv_y/v_z respectively.

This scaling will capture the effect of perspective as can be seen in Figure 2. Despite the two Cobblemon having the same height in View / World Space, their projected heights onto the viewing / near plane (measured from y = 0 to the projection point) is such that the further cobblemon is smaller than the closer cobblemon.

Now, we just need to make sure that points inside the view frustum are mapped to normalized device coordinates. That is, points that intersect the top plane are mapped to y = 1, bottom y = -1, left x = -1, right x = 1.

To do that, let's now imagine that we are the camera and are looking at the view port.



So we can now determine the first 2 rows of the matrix. To scale the x and y components by n/v_z, we need to set $m11$ and $m22$ to be n and everythign else in rows 1 and 2 to be 0.

$$\mathbf{Pv_{view}} = \begin{pmatrix} \text{n/z} & 0 & 0 & 0 \\ 0 & \text{n/z} & 0 & 0 \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \\ z_{view} \end{pmatrix}$$

From the maths, we can immediately see that the apprent size of the image on the near / viewing plane is inversely proportional to its distance from the viewer, and directly proportional to the distance from the plane to the viewer.

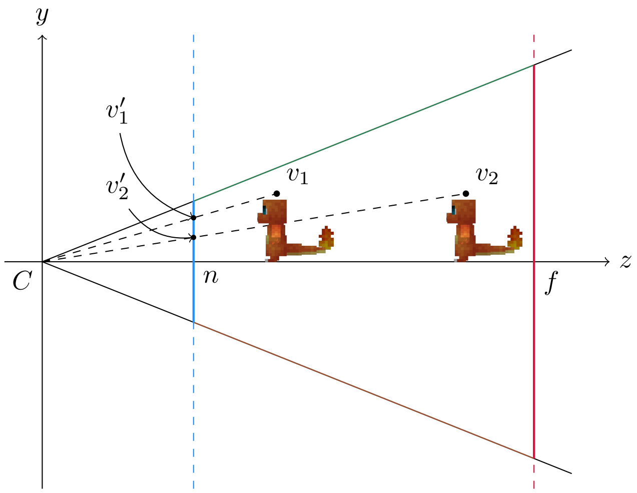
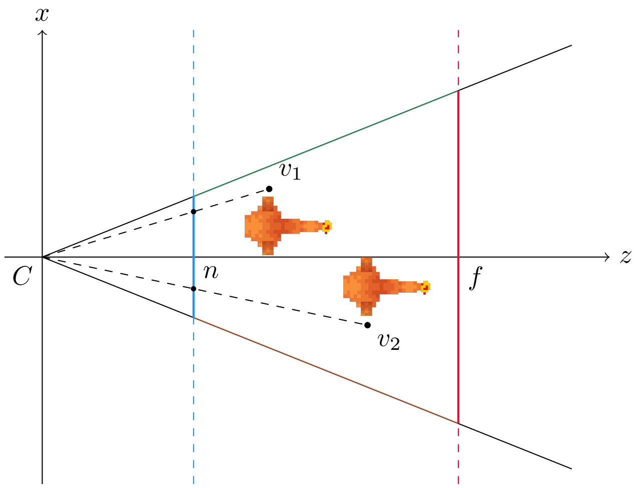

View Space

Figure 2: 2D slice of the view frustum along the yz plane depicting the top, bottom, near and far planes. The vertex v is projected to the point v' on the near plane by calculating the intersection between a line drawn from the origin to v and the near plane. The Camera / Viewer, labelled C is at the origin, thus this Coordinate Space is View Space.


View Space

This transformation on the x and y coordinates is actually enough to capture the effect of perspective. However, since we essential squish everything in the View Volume onto a 2D plane, unless we do something about it, we lose out on Depth information.

Depth is important to us for two reasons:

1. Relative ordering of objects in the Scene for depth test
2. View Space depth for Shader calculations / Alpha blending

To solve 1, we come up with some transformation of the z value to map it to our Canonical View Volume which, in my case, is 0 - 1. We require that vertices on the near plane are mapped to 0 and vertices on the far plane are mapped to 1.

$$\mathbf{Pv} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

5.5 Deriving The Viewport Matrix

5.6 The Vertex Shader

5.7 Clipping & The Sutherland Hodgman Algorithm

6 Drawing to the screen

6.1 What is *Rasterization*?

6.2 Interpolation

6.3 Rasterizing Triangles

6.4 What is a *Fragment?*

6.5 The Fragment Shader

6.6 Depth Testing

6.7 The Framebuffer and Z-Buffer

7 Shading

7.1 Shaders

7.2 Materials

- A Material
- It contains ...

7.3 Textures

7.4 Unlit Shader

7.5 Lit Shader

7.6 Phong Shader

7.7 Toon Shader

7.8 Shadows

7.9 Ambient Occlusion

8 Appendix

8.1 Quaternions

8.2 Barycentric Coordinates

9 Deprecated

9.1 Math Concepts

9.1.1 Vectors

- In this document, we will use typical Vector operations and notation including but not limited to the following
- A vector is $\mathbf{v}$ 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

9.1.2 Planes

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

9.1.3 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)

9.1.4 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.

9.1.5 Coordinate Spaces

- Cartesian
- Projective
- Barycentric
- Spherical, Cylindrical

9.1.6 Transformations

- Model, View, Projection

9.1.7 Deriving the Model Matrix

9.1.8 Deriving the View Matrix

9.1.9 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 ?. 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.

9.1.10 Deriving the Viewport Matrix

9.2 Algorithms

9.2.1 Rasterization

9.2.2 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****

9.2.3 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

10 Conclusion & Future Work

10.1 What I learnt

10.2 Limitations

10.3 A future implementation in C++

11 References