CPU Graphics Engine

Thomas Dizon

December 12, 2025

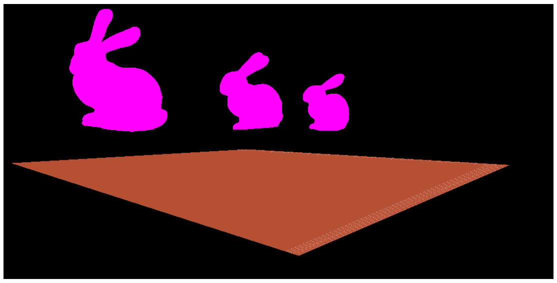


Figure 1

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 The Vertex Shader 7
5.5 Perspective Projection 7
  5.5.1 What is *Perspective*? 8
  5.5.2 The View Frustum 8
  5.5.3 Deriving the Projection Matrix 9
  5.5.4 Perspective Divide 12
5.6 Deriving The Viewport Matrix 12
5.7 Clipping & The Sutherland Hodgman Algorithm 12

6 Drawing to the screen **12**
6.1 What is *Rasterization*? 12
6.2 Interpolation 13
6.3 Rasterizing Triangles 13
6.4 What is a *Fragment?* 13
6.5 The Fragment Shader 13
6.6 Depth Testing 13
6.7 The Framebuffer and Z-Buffer 13

7 Shading **13**
7.1 Shaders 13
7.2 Materials 13
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 14
  9.1.3 Matrices 14
  9.1.4 Quaternions 14
  9.1.5 Coordinate Spaces 14
  9.1.6 Transformations 15
  9.1.7 Deriving the Model Matrix 15
  9.1.8 Deriving the View Matrix 15
  9.1.9 Deriving the Projection Matrix 15
  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 The Vertex Shader

5.5 Perspective Projection

- In a sentence, the goal of the Perspective Projection is to map the View Space ‘View Frustum’ to ‘Canonical View Volume’ in such a way that perspective is simulated.
- The Canonical View Volume is 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 and a format understood by our rasterizer.
- Perspective Projection is broken up into two steps:
  1. Projection Matrix
  2. Perspective Divide

- The *Projection Matrix* transforms a vertex from View Space into Clip Space and the *Perspective Divide*, which is ultimately responsible for simulating perspective, is given by the following operation:

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \to \begin{pmatrix} x/w \\ y/w \\ z/w \\ 1 \end{pmatrix}$$

- If you are confused, please keep reading! I go into more detail here.

5.5.1 What is *Perspective*?

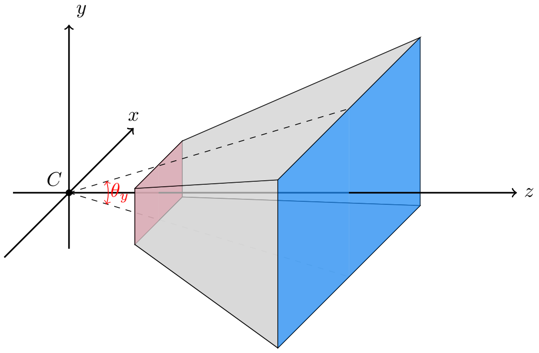
- In this project, my goal is to create graphics that are visually realistic or at least somewhat consistent with our experiences in the real world. Thus, my definition of Perspective will be made with that in mind.
- We start by assuming the viewer is 'point like'. We, as humans, perceive the world through our eyes which are 'point-like', in contrast to something like a Photosensor which perceives the world through a rectangular array of sensors.
- Put simply, Perspective is the phenomenon of objects appearing smaller to a point-like viewer as it moves further away from the viewer.
- In other words, the apparent size of an object to a viewer is inversely proportional from the object's distance to the viewer. This is a direct consequence of basic trigonometry.

5.5.2 The View Frustum

- In this section, we motivate the definition of a *View Frustum* and define what it is.
- To begin, I want you to close one eye and hold your index finger in front of it, close enough for it to appear blurry. Then, gradually move your index finger further away until it comes into focus. We define the distance between your eyeball and finger to be n, the *near plane* distance.
- In our renderer, we choose not to render objects between the viewer and the near plane.
- Pause for a moment. Take your right hand and point straight ahead with your index finger. Slowly move your arm as high as you can until the tip of your index finger is just invisible. Now do the same with your left arm. The angle between your arms is your Vertical Field of View (FOV) or θ_y. You can easily repeat the exercise going left and right.
- This angle essential defines the edges of your vision. So it will be an important quantity for us in trying to recreate a realistic viewer.

- Since we are trying to create realistic graphics on a computer screen, we need to take that screen into account. So the last quantity that will be important to us is the *aspect ratio* which is the ratio between the width and height of the widow you're using to render this world.
- We also don't want to render infinitely into the distance because of the limited memory on our computers, so we also define f the distance of the *far plane* from the Camera along the z axis.
- So what objects are we going to render?
- We start with the near plane. We use the FOV θ_y to carve out a rectangular area that extends infintiely in the x direction. Then we use a to figure out the distance in the x dimension the near plane should be limtied to.
- Then, we follow the line until we reach $z = f$ and connect all the points to construc the view frustum.

View Space



5.5.3 Deriving the Projection Matrix

- In this section we derive the Projection Matrix which maps the View Frustum to homogeneous Clip Space.
- We break it up into four steps
    1. Scale x and y components such that perspective is captured.
    2. Scale x and y components again to $[-1, 1]$ range
    3. Map z to $[0, 1]$ range such that relative depth is preserved
    4. Store original view space depth in w variable

1. Scale x and y such that perspective is captured

- To capture perspective, we project each vertex v on the near plane via a line that passes through the origin and v.
- To project a vertex $\mathbf{v}$ onto the near plane, we draw a line from the origin to that point and mark where it intersects the near plane.
- 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, $\mathbf{A} = (0, 0, 0)$ and $\mathbf{B} = \mathbf{v}$. Thus,

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

  We know that the z-component of the intersection point is going to be n.

  $$\mathbf{r}(t_i) = (v_{xt_i}, v_{yt_i}, v_{zt_i}) = (v_{xt_i}, v_{yt_i}, n)$$

  Equating the z-components:

  $$t_i = n/v_z$$

  Thus,

  $$\mathbf{r}(t_i) = \left(\frac{nv_x}{v_z}, \frac{nv_y}{v_z}, n\right)$$

- 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.

View Space

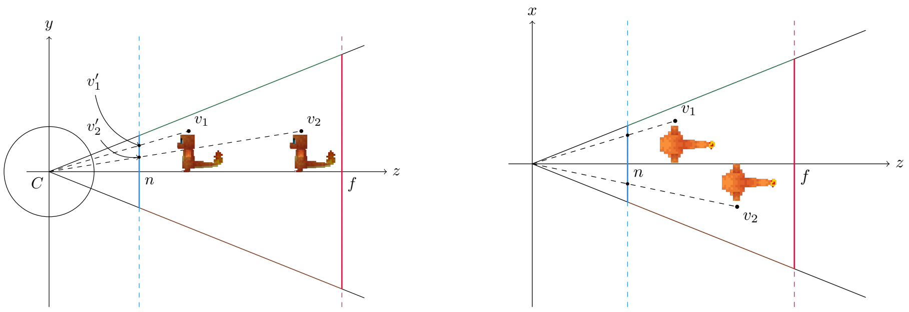


Figure 2: Side by side graps of the x-z slice and y-z slice of the view frustum showing the how the projected image is calculated on the near plane

2. Scale x and y to $[-1, 1]$ range

- 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 this, let the distance from the z axis to the top of the viewport be r and the distance from the z axis to right side of the viewport be r.

- To match the dimensions of our window, we define the aspect ratioo a to be $a = r/t$

- We can calculate r by considering FIGURE X. $\tan(\frac{\theta_u}{2}) = \frac{r}{n}$ so then $r = ntan(\frac{\theta_y}{2})$.

- Then since $a = r/t$, $t = r/a$ which means $t = \frac{ntan(\frac{\theta_y}{2})}{a}$

- Now, if we scale the y component of all vertices on the near plane by $1/t$, this means that vertices in the range $[-r, r]$ will map to $[-1, 1]$ and if we scale all the x components by $1/r$ all of the Vertices' x components will be in the range $[-1, 1]$ which is exa tly what we need.

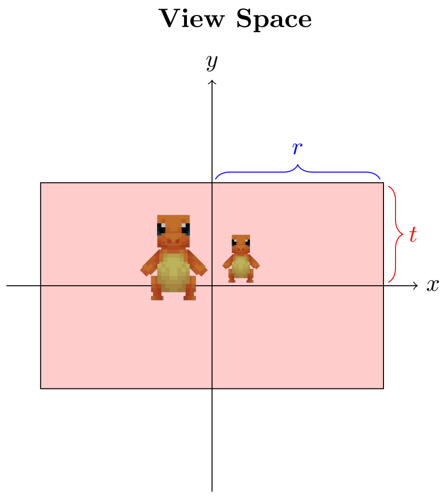


Figure 3: The view frustum from the perspective of the Camera

- We now have what we need to figure out the firs two rows of the projection matrix. We need to scale the x component by **TODO** and the y component by **TODO** which can be done with the following matrix:

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

3. Scale z to $[0, 1]$ range

- In order to determine the matrix elements that will transform the input v_z into the desired $[0, 1]$ range, we work backwards by considering what elements would be needed to map a point on the near plane to $z = 0$ and a point on the far plane to $z = 1$
- We would need

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ n \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 0 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ f \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \\ 1 \end{pmatrix}$$

- We would need m_{33} and m_{34} to be **TODO** and **TODO** respectively
- How we do this mapping doesn't actually matter that much, as long as relative ordering is preserved.

4. Store w value

- Simply set m_{43} to be 1
- Talk about homogeneous coordinates

The Final Matrix

- Combining all the steps together, the final matrix is given by:

$$\mathbf{P} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

5.5.4 Perspective Divide

- Once we have applied the perspective

5.6 Deriving The Viewport Matrix

5.7 Clipping & The Sutherland Hodgman Algorithm

6 Drawing to the screen

6.1 What is *Rasterization*?

Hello

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 **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 3×3 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.
- 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