The 3D Graphics Pipeline, in Pure C

Thomas Dizon (@tommiedevelops)

March 1, 2026

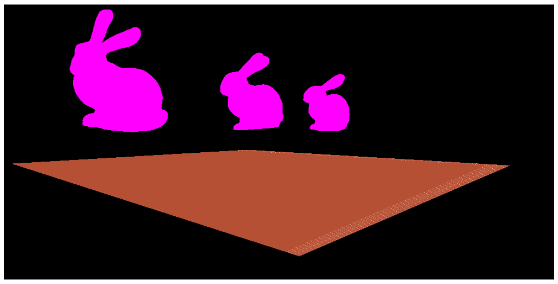

Figure 1

Contents

1 Introduction 4
1.1 Motivation 4
1.2 The scope of a *Graphics Engine* 4
1.3 Why *C*? 4
1.4 What does the project depend on? 5

2 System Design 5
2.1 Diagram 5
2.2 Overview 5

3 3D Geometry in Memory 5

4 Coordinate Spaces 6
4.1 What is a *Coordinate Space*? 6
4.2 GameObjects & Object Space 6
4.3 Scenes & World Space 6
4.4 Cameras & View Space 6

5 Transforming Vertices 6
5.1 Matrices as Transformations 6
5.2 Deriving the Model Matrix 6
5.2.1 Transforms 6
5.2.2 Scale Matrix 6
5.2.3 Rotation Matrix 6
5.2.4 Translation Matrix 6
5.2.5 Combining them together 6
5.3 Deriving the View Matrix 6
5.3.1 The Matrix Inverse 6
5.3.2 Inverse of the Model Matrix 6
5.4 The Vertex Shader 6
5.5 Perspective Projection 6
5.5.1 What is *Perspective*? 7
5.5.2 The View Frustum 7
5.5.3 Deriving the Projection Matrix 8
5.5.4 Perspective Divide 11
5.6 Deriving The Viewport Matrix 11
5.7 Clipping & The Sutherland Hodgman Algorithm 11

6 Drawing to the screen 11
6.1 What is *Rasterization*? 11
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 12
7.4 Unlit Shader 12
7.5 Lit Shader 12
7.6 Phong Shader 12
7.7 Toon Shader 12
7.8 Shadows 12
7.9 Ambient Occlusion 12

8 Appendix **12**
8.1 Quaternions 12
8.2 Barycentric Coordinates 12

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 The scope of 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*?

- C is an extremely tedious language to program in as it lacks many modern features like Object Oriented Programming which makes scaling a program much easier and less time consuming.

- While I generally would not reccomend programming in C for serious applications, it is a great tool to maintain a precise mental model of your code as you are maanging every bit of memory manually.
- My rule of thumb is, if you want to undertand how X works, you should implement a basic version of X in C as it forces you to understand every detail of how it works.
- Then, for a serious implementation of X, you should use a higher level language like C++ or C# so it takes you far less time.
- As such, I have developed this project in C to gain a deep understanding of the workings of graphics systems so that I may use this knowledge in my future higher level projects.

1.4 What does the project depend on?

- Right now, it depends on the C standard library, std_image.h for dealing with .png files and SDL2 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 System Design

2.1 Diagram

2.2 Overview

- Scene Manager
- Asset Manager
- Renderer
- Game Math
- Application
- Platform
- Third Party

3 3D Geometry in Memory

- In computer memory, 3D objects are represented as a collection of vertices and triangles. A vertex is a point in 3D space defined by its x, y and z coordinates. A triangle is a primitive that consists of three vertices and represents a flat surface in 3D space. By connecting multiple triangles together, we can create complex 3D shapes and models. This method of representing 3D geometry is efficient and widely used in computer graphics, allowing for the creation of detailed and realistic virtual environments.

- In this project, I store 3D objects as Wavefront .obj files.
- I have a dedicated .obj parser which converts a .obj file into a 3D Mesh in program memory.
-

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 to 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 the *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. Thus the last quantity that will be important to us is the *aspect ratio*, a, which is the ratio between the width and height of the window you're using to render the virtual 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.
- In summary, the set of objects we care about rendering are those found within the volume in Figure 2.

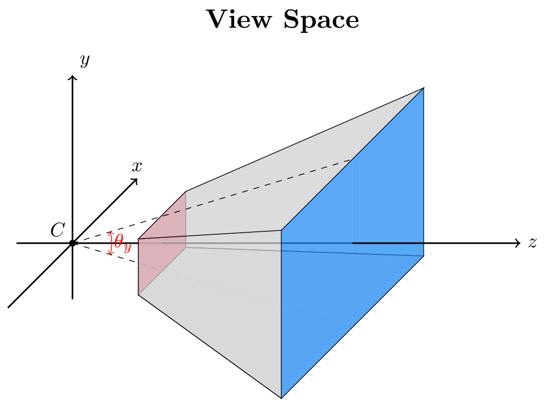

Figure 2: The View Frustum in View Space

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

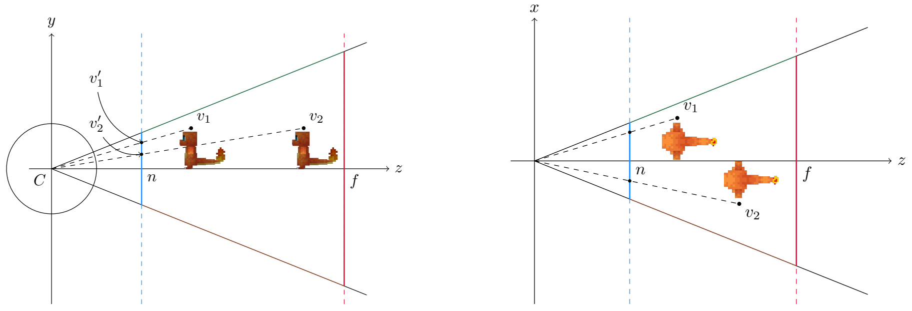

Figure 3: Side by side graps of the x-z slice and y-z slice of the view frustum showing the how the projected image is calculated on the near plane

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.

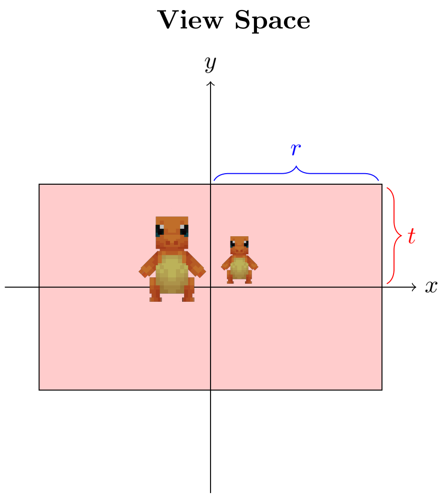

Figure 4: The view frustum from the perspective of the Camera

   - 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