CPU Graphics Engine

Thomas Dizon

November 1, 2025

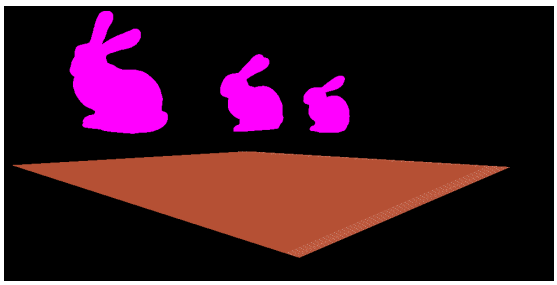

Figure 1:

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.

1.1.1 What is a *Software Rasterizer*?

- 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)

1.1.2 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

1.1.3 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 Background

2.1 Math Concepts

2.1.1 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

2.1.2 Planes

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

2.1.3 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)

2.1.4 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.

2.1.5 Coordinate Spaces

- Cartesian
- Projective
- Barycentric
- Spherical, Cylindrical

2.1.6 Transformations

- Model, View
- Projection
- Viewport

2.2 Algorithms

2.2.1 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****

2.2.2 Clipping

- This algorithm is used after the Vertex Shader when Vertex positions are in Clip Space. The goal of the algorithm is to determine which Vertices are outside of the Camera's view frustum, and clip them. If a Triangle contains some vertices inside and some outisde, it also has to reconstruct the Triangle using the edge of the frustum.
- ****TODO**** Write some pseudocode, include a diagram

3 System Architecture

3.1 Overview

3.1.1 System Diagram

3.1.2 Project Structure

3.1.3 Conventions

3.2 Data Design

3.2.1 Primitives: Triangles and Vertices

- In this project, a Triangle consists of an array of size 3 of VSout structures which is the output of the Vertex Shader. This is, in essence, the same thing as a Vertex.
- The VSout consists of Vec4f clip space position, Vec2f uv coordinates,Vec3f normals along with some other stuff. ****FIX!!****

3.2.2 Meshes and Materials

- A mesh is a collection of Vertex data in the form of a Vector array and corresponding *Index Arrays* which tells us which Triangles the data belongs to. For example in FIGURE X.
- A Material is a container for various properties like Albedo, Specular Intensity, etc. which tells the Renderer how to render the geometry. Is also contains a reference to a Pipeline structure which consists of a Vertex and Fragment shader.

3.2.3 Framebuffer and Zbuffer

- The FrameBuffer structure contains a 1D uint32_t array of size $ScreenWidth*$ $ScreenHeight$ where the result of pixel (x, y) on the screen is stored in $ScreenWidth * y + x$ in the array.
- It also stores a 1D float array of the same dimensions which stores the depth value of the closest fragment to the screen at that pixel coordinate

3.2.4 Scene, GameObject, Camera, Light

- Scene stores a Dynamic Array of GameObjects, a single Camera and a single Light source. The scene is the home of World Coordinates where you can say how different objects in the scene are placed relative to each other.
- GameObject consists of a Transform which tells us where its Position, Rotation and Scale in World Space, a Mesh which tells us the geometry of the object and a Material which tells us how to render the object.

GameObject is the home of Model/Object space coordinates which is used by its Mesh to detail its Geometry.

- Camera contains all the parameters to specify the View frustrum which is used to decide what objects can be rendered to the screen. It also has a Transform. Camera is the home of View space coordinates which is used as part of the Render Pipeline.
- Light is a simple directional light. So it contains a Vec3f for its direction in world space and a Vec4f for its color.

3.3 Software Design

3.3.1 Procedural Programming vs. Object Oriented Programming

* I decided to learn towards a Procedural Programming approach in this project because of the language of choice (C) * This ended up being a limitation because when I wanted to make the system more complex, it was difficult to refactor and extend

3.3.2 Graphics Pipeline

3.3.3 Scene Management

3.3.4 Rendering

3.3.5 Shading

4 Conclusion & Future Work

4.1 What I learnt

4.2 Limitations

4.3 A future implementation in C++

5 References