Miles Rollins-Waterman

Introduction to Computer Graphics

Prof. Hanocka

January 28, 2026

VIEWPORT MATRIX:

The section was relatively simple. I copied the given Mvp matrix into a numpy array to construct the viewport matrix. Then for each face in the obj, I found its vertices and converted them to screen coordinates by multiplying them by the viewport matrix I constructed. Finally, I colored each screen coordinate vertex with the color of the face it's associated with.

ORTHOGRAPHIC PROJECTION:

First I constructed the orthographic matrix as described in the textbook. Next I found the vertices of each face, just like with the viewport matrix, but instead of converting them directly, I first found the bounding box for each face from its vertices. I converted these bounding boxes to screen coordinates using the orthographic transformation matrix and then iterated over them. For each screen coordinate (i.e pixel) in the bounds of each face's bounding box, I found the center of the pixel and checked whether it was in that face/triangle using barycentric coordinates. Finally I colored each pixel within the triangle/face, ignoring ones that weren't.

CAMERA PROJECTION:

Camera projection followed the same process as the previous two (construct the matrix, find the bounding box, convert to screen coordinates, check each pixel in the bounding box for coverage, etc) with a notable difference in how world coordinates were converted to screen coordinates. Previously, I had been constructing a "dummy" matrix to hold the x and y values of the world coordinate to be translated in order to match the shape required to multiply by the transformation matrix. This matrix was essentially x_w, y_w, 1, 1. This worked for the orthographic and viewport transformation since we did not care about the z-axis, but the camera projection required more. Instead of filling the matrix with 1,1 after the world coordinates I filled it with z_w, 1 to include the z-axis information in the transformation to screen coordinates.

I referenced this OpenGL tutorial to understand the camera transform better, and figure out how to turn the 3D lookat point into a gaze vector that could be used to construct the matrix as shown in the textbook.

PERSPECTIVE PROJECTION:

Similar to the camera, the perspective projection followed the same broad steps as previous parts but differed in its coordinate conversion function. Now the conversion function required not only the z-axis but also the w variable of the world coordinate. So the world coordinate matrix now looked like x_w, y_w, z_w, w_w. This matrix was then multiplied by the perspective transformation matrix (as constructed in the textbook) but before returning the screen coordinates, I applied the perspective divide.

Z-BUFFER & COLOR INTERPOLATION:

First, I implemented the color interpolation. For each pixel covered by a triangle, I found its barycentric coordinates. Next I found the color value at each vertex of the face that pixel was on/covered by. Finally, I interpolated the color value at that specific pixel using its barycentric coordinates and the vertex colors (as per the slides for lecture 5). The z-buffer was a very similar process. For each covered pixel, I found the z-axis values of each vertex of the face that covered it and used those z-axis values alongside the pixel's barycentric coordinates to interpolate a depth value for that pixel on that face. In order to have each pixel track its depth value, I created a z-buffer matrix the same size as the img array, but with only 1 value at each point. These values were all initialized to infinity at the start of the render program, and every time a pixel might need to be drawn (i.e. was covered by a triangle) its depth was interpolated and checked against the z-buffer entry corresponding to that pixel.

BIG SCENE:

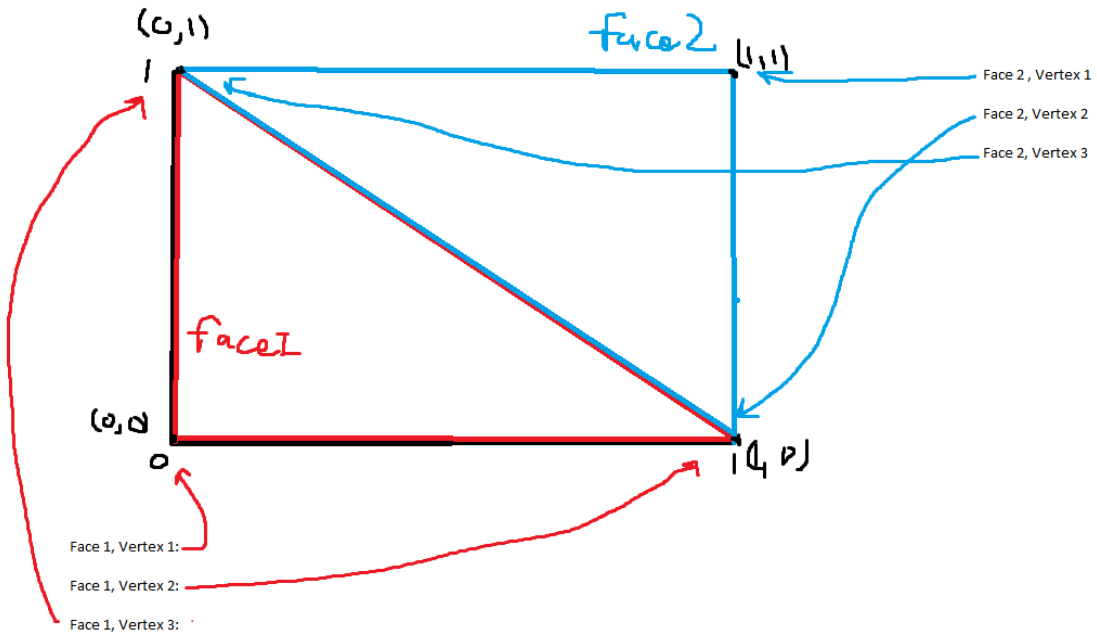
This was incredibly straight forward, I changed no code from render_zbuffer, except that I modified the camera variables as per the requirements and added an extra layer to the for loop so that the render function would check every face of every object provided in the list.

TEXTURE MAPPING:

This was much the same as the z-buffer / color interpolation, but the color for the pixel was calculated differently. Instead of taking the 3 vertex colors for the pixel's face. The new process was as follows: find the (u,v) coordinates of each vertex for the current face. Using the barycentric coordinates of the given pixel, and the *w* value of the vertices, apply the perspective

correction as per the lecture slides from lesson 05 and interpolate the (u,v) coordinates of the given pixel. Finally, clamp those (u,v) coordinates as per the textbook and grab the color value from the texture map/image at that (u,v) coordinate. Depth interpolation is accounted for in the same way as with the z-buffer part, and the pixel is drawn or not accordingly.

The u,v coordinates for a given face were determined by pairing up each face with the one after it (a pair in this case represents one full side of the cube), then assigning each vertex of those faces a "corner" of the u,v graph. See the diagram below for details:



The interpolated u,v coordinate was multiplied by the texture's width and height respectively (clamping to the nearest neighbor) in order to map it to a real pixel on the texture image.

MATH QUESTIONS:

1a. The orthographic view volume is along the negative z-axis so far values should be more negative than near ones hence n > f.

1b.

1. $x_{canconical} = \text{scaleX} \cdot (x - \text{centerX})$

$$\text{centerX} = \frac{ortho_{left} + ortho_{right}}{2}$$

$$\text{scaleX} = \frac{2}{ortho_{right} - ortho_{left}}$$

$$x_{canconical} = \text{scaleX} \cdot (x - \text{centerX}) \Rightarrow x \cdot \frac{2}{ortho_{right} - ortho_{left}} - \frac{ortho_{right} + ortho_{left}}{ortho_{right} - ortho_{left}}$$

2. $y_{canconical} = \text{scaleY} \cdot (y - \text{centerY})$

$$\text{centerY} = \frac{ortho_{bottom} + ortho_{top}}{2}$$

$$\text{scaleY} = \frac{2}{ortho_{top} - ortho_{bottom}}$$

$$y_{canconical} = \text{scaleY} \cdot (y - \text{centerY}) \Rightarrow y \cdot \frac{2}{ortho_{top} - ortho_{bottom}} - \frac{ortho_{top} + ortho_{bottom}}{ortho_{top} - ortho_{bottom}}$$

3. $z_{canconical} = \text{scaleZ} \cdot (z - \text{centerZ})$

$$\text{centerZ} = \frac{-ortho_{near} + (-ortho_{far})}{2}$$

$$\text{scaleZ} = \frac{2}{ortho_{far} - ortho_{near}}$$

$$y_{canconical} = \text{scaleZ} \cdot (z - \text{centerZ}) \Rightarrow z \cdot \frac{2}{ortho_{near} - ortho_{far}} - \frac{ortho_{near} + ortho_{far}}{ortho_{near} - ortho_{far}}$$

2. The following is true if $z_1 > z_2$:

Solution: For z_1, z_2 in canonical coordinates, let z_1' be the value of z_1 after applying the P matrix, and the same for z_2. Then:

$$z_1' = n + f - \frac{fn}{z_1'} \quad (1)$$

$$z_2' = n + f - \frac{fn}{z_2'} \quad (2)$$

$$z_1' > z_2' \Rightarrow n + f - \frac{fn}{z_1'} > n + f - \frac{fn}{z_2'} \quad (3)$$

$$(n + f) - n + f - \frac{fn}{z_1'} > (n + f) - n + f - \frac{fn}{z_2'} \rightarrow \quad (4)$$

$$\frac{fn}{z_1'} > \frac{fn}{z_2'} \Rightarrow \frac{1}{fn} \cdot \frac{fn}{z_1'} > \frac{1}{fn} \cdot \frac{fn}{z_2'} \quad (5)$$

$$z_1' > z_2' \quad (6)$$