Computer Graphics Using OpenGL

Yu Han Yang (andyyhy), Jason Liu (jyliuu),
Quintin Dwight(qdwight), Yijun Li (junnnli)

Abstract

Math 214 is an introductory course to linear algebra and covers matrix manipulation concepts used in many real-world applications. We will be presenting the applications of linear algebra in 3D computer graphics using the OpenGL library in C++.

Introduction

OpenGL is a library that provides us with a large set of functions that can be used to manipulate graphics and images. Graphic libraries such as OpenGL render 3D information onto a 2D screen through various matrix operations. We will create a program capable of rendering objects that have different positions, rotations, and sizes. In our demo, we will be using the Utah teapot, which is a commonly used object in computer graphics.

Computer Graphics

Computer graphics is a broad and diverse field. Today, the high-quality graphic displays in personal computers provide one of the most natural ways of interacting with a computer. It deals with the entire process of creating computer-generated imagery, from creating digital three-dimensional models to the process of texturing, rendering and lighting the models. This ability to render 3D information onto a 2D screen significantly enhances the ability to understand information, perceive trends, and visualize real or imaginary objects either moving or stationary in a close to real-world environment.

Real World Applications

Computer graphics is crucial in all sorts of applications nowadays such as movie making, photo editing, scientific modeling and much more. For example, in automotive engineering the ability to quickly visualize newly designed shapes or components is indispensable. Before the advent of computer graphics, designers built expensive prototypes and time-consuming clay models to visualize their ideas. However, with advancements in computer graphics, designers can interactively view and modify 3D models of their prototypes using a computer, significantly reducing R&D costs.

Utah Teapot

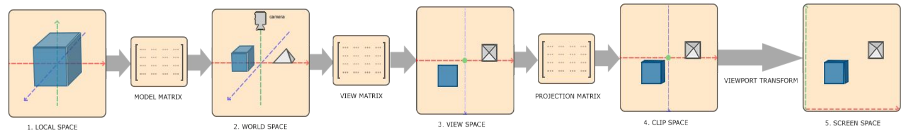
GitHub Repository: https://github.com/JasonLiu2002/math214-opengl

To apply what we've learnt about linear algebra to computer graphics, we have rendered and shaded the famous Utah Teapot using the C++ OpenGL library. We will cover the different spaces OpenGL uses, its Model View Projection (MVP) matrix, camera manipulation, and the mathematical operations used.

Coordinate Systems

In order to model the 3D object on the 2D screen we see, the object's vertices are transformed through a variety of coordinate systems. The advantage of transforming them to several intermediate coordinate systems is that some calculations or operations are easier in certain coordinates.

1. Local Space: Coordinates of a model's vertices relative to its individual origin
2. World Space: These coordinates are with respect to some global origin of the world, which contains a variety of models and other objects
3. View Space: Coordinates with respect to where the camera is and where it is looking.
4. Clip Space: Homogeneous coordinates. "Clip" refers to discarded vertices that are not within the frustum of the camera.
5. Screen Space: Final 2D coordinates on screen



Model Matrix

Once we have a set of vertices in the local space defining an object, we "place" it in the world space through the model matrix transformation. By scaling, rotating, and translating the model, we can place it in the world space in any size, orientation, and position.

$$\text{Model Transformation Matrix} = \text{Translation Matrix} * \text{Rotation Matrix} * \text{Scale Matrix}$$

$$\text{Translation Matrix} \Downarrow \begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation Matrix ⇓ To properly implement rotations, quaternions are used which is outside the scope of this project.

$$\text{Scale Matrix} \Downarrow \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

View Matrix

The view matrix maps a given vertex from world space to camera space. R, U, and D are the right, up, and direction vectors respectively of the camera. They form an orthonormal basis for $\mathbb{R}^3$. In computer graphics, instead of moving the camera around the scene, we translate all other objects around the camera. This is why R, U, D are transposed and why P is negated.

When representing a camera, we only keep track of the U, D, and P vectors. The R vector is obtained by crossing the U vector with the D vector.

$$V = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Projection Matrix

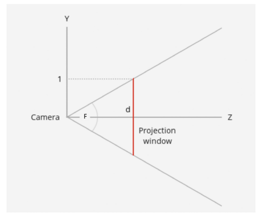
The projection matrix maps a given vertex into clip space where:

Vertex: 4 tuple (x, y, z, w) representing a point in 3d space. The w component is always 1 for points and 0 for directions.
Clip Space: 4 tuple (x, y, z, w) representing a point in 3d space. The w component now represents depth; w is larger for objects that are farther away from the camera. Eventually to obtain the 3 tuple position x, y, and z will be divided by w. Vertices with z-component outside of [-1, 1] will be discarded, or "clipped."

$$P = \begin{bmatrix} \frac{1}{a*\tan(\frac{F}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{F}{2})} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{2*f*n}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Here, P is the projection matrix, F is y-axis fov, a is the screen ratio, f is far z-axis far clipping, n is z-axis near clipping. These values can be imagined to create a frustum where points outside the defined volume will be clipped.

Derivation



$$\frac{1}{d} = \tan(\frac{F}{2}) \implies d = \frac{1}{\tan(\frac{F}{2})}$$

$$\frac{p_x}{d} = \frac{v_x}{v_z} \implies p_x = \frac{v_x * d}{v_z} = \frac{v_x}{v_z * \tan(\frac{F}{2})}$$

$$\frac{p_y}{d} = \frac{v_y}{v_z} \implies p_y = \frac{v_y * d}{v_z} = \frac{v_y}{v_z * \tan(\frac{F}{2})}$$

Let v be a vertex viewed through the red projection window shown above (only yz-plane shown) transformed into a new vector p. Notice that v_z is in the denominator. OpenGL automatically divides all other components by v_w for us. We set v_w to $-v_z$ via the -1 in P (why this is negative is an irrelevant OpenGL detail). Therefore we do not need to include $1/v_z$ in the matrix.

We find $P_{3,3} = -(f+n)/(f-n)$ and $P_{3,4} = 2(f)(n)/(n-f)$ by satisfying the equation $f(z) = P_{3,3} + P_{3,4}/v_z$ with constraints $P_{3,3}+P_{3,4}/n = -1$ and $P_{3,3}+P_{3,4}/f = 1$. This maps the z-coordinates of v in [n, f] to [-1, 1], which are called normalized device coordinates. All other vertices are thrown out as they are not visible to the camera.

Conceptually speaking, the projection matrix applies a scale that corresponds to a field of view. We also keep track of depth, culling all values that do not fall within a desired near and far threshold. It is the final step before converting to screen coordinates. Screen coordinates are then sent off to the fragment shader in order to determine pixel colors.

Works Cited: Bretscher, Otto. *Linear Algebra with Applications*. Pearson Education, 2013., https://nautil.us/blog/the-most-important-object-in-computer-graphics-history-is-this-teapot, https://learnopengl.com, https://open.gl, http://www.opengl-tutorial.org, https://www.opengl.org, https://www.khronos.org/opengl