Homogeneous Coordinates And Its Applications in Computer Graphics

Mayaank Ashok (2022111022)
Aniruth Suresh (2022102055)
Sri Vishnu Varun (2022102031)

June 11, 2023

1 Introduction

The ultimate goal of Computer Graphics is to display objects and shapes on a 2D screen. Two of the most popular methods of rendering objects are Rasterization and Ray Tracing. While the algorithms in ray tracing are rich in linear algebra and vector algebra in their own right, we will focus our attention on the application of linear algebra in Rasterization.

The most prevalent representation of 3D models in software is by dividing the surface of the model into triangles, called a mesh, and then specifying the coordinates of each vertex , and how the vertices are connected to form triangles

In such representations, the vertices of the object are described with a coordinate system originating at the center of the object. These coordinates are known as 'Object Coordinates'. In a normal render, we have multiple objects in the world each at a different position and a single camera. The 2D screen of the Computer is assumed to be attached to the camera thus the position of the objects on the screen are also determined by the orientation of the camera.

We assume there exists an origin point in the world that provides a reference point for the coordinates of all objects and the camera. The position of the vertices of each object with respect to the world origin are known as 'World Coordinates'.

Further, the 'Eye Coordinates' of an object is its position from the perspective of the camera, where the viewer is looking in the negative z axis, and the x and y axis point rightwards and straight up respectively.

The Eye Coordinates correspond closely to the objects position on the screen, and the Projection Transform converts the Eye Coordinates to coordinates where all objects within $(-1, -1)$ to $(1, 1)$ are drawn on the screen (i.e 'viewport'). This is therefore known as the 'Viewport Coordinates'

The sequence of transforms that convert 3D coordinates of an object into its 2D position on the screen is shown below.

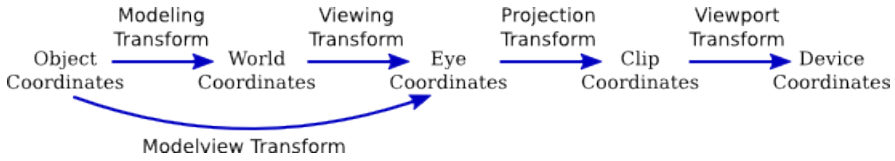

Figure 1: Graphics Pipeline

2 Representing Transforms as Matrix Multiplications

As mentioned before, we can represent the position of any point in 3D space as a vector of the form $\bar{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$. It can also be further noted that all transformations mentioned in the previous section can be represented as composition of Scale, Rotation, and Translation transformations.

1. Scale Matrix

   The scale transformation uniformly scales each coordinate of a vector by given factor. Thus the scale transformation maps (x, y, z) to $(s_x x, s_y y, s_z z)$. Where s_x, s_y, s_z are the scale factors in the x, y, z axes respectively. We can represent this transformation as a matrix multiplication:

   $$S\bar{v} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \end{bmatrix}$$

2. Rotation Matrix

   The rotation transform $R_z(\theta)$ rotates a vector by θ counter-clockwise about the z axis.

   Therefore the vector $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$ gets mapped to $\begin{bmatrix} x\cos(\theta) - y\sin(\theta) \\ x\sin(\theta) + y\cos(\theta) \\ z \end{bmatrix}$ We can represent this transformation as a matrix multiplicatio:

   $$R_z(\theta)\bar{v} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x\cos(\theta) - y\sin(\theta) \\ x\sin(\theta) + y\cos(\theta) \\ z \end{bmatrix}$$

   Similarly we can write the matrices for rotation about x and y axes, $R_x(\theta)$ and $R_y(\theta)$ respectively. Any rotation in 3D space can be write as a matrix multiplication of the 3 axis oriented rotations.
   In general, $R = R_z(\gamma)R_y(\beta)R_x(\alpha)$ represents an extrinsic rotation with Euler angles α, β, γ about the axes x, y, z respectively.

3. Translation Matrix
   The Translation transformation adds a translation component $(\Delta x, \Delta y, \Delta z)$ to every vector. Thus (x, y, z) gets mapped to $(x+\Delta x,\ y+\Delta y,\ z+\Delta z)$.

However it is clear that such a transformation, T, cannot be represented as a matrix multiplication on a vector as $T \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$ will hold for any matrix T irrespective of translation components $(\Delta x, \Delta y, \Delta z)$. This is clearly not desired behaviour.

3 Homogeneous Coordinates

Homogeneous Coordinates is a system of coordinates that is designed to operate in the Projective Plane. The projective plane is an extension of the Euclidean Plane that includes a 'line at infinity' that can be represented in finite coordinates. Homogeneous coordinates also provide an elegent way of representing families of straight lines and conic sections.

Mobius in his 1827 work introduced this concept by appending an extra coordinate, such that the point (x, y, w) in homogeneous coordinates has a projection onto the 2D Euclidean plane at $(\frac{x}{w}, \frac{y}{w})$.

Similarly we can represent any point in 3D space as the vector $\bar{v} = (x, y, z, 1)$

4 Translation and Shear Matrices using Homogeneous Coordinates

In 2D coordinates, a vector is represented as $\bar{v} = (x, y, 1)$ in homogeneous coordinates. We can apply a shear along the z-axis using matrix multiplication:

$$\begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + \Delta x \\ y + \Delta y \\ 1 \end{bmatrix}$$

This matrix shears a unit cube Δx units along the x-axis and Δy units in the y-axis. However, we can notice that the resultant vector is the homogeneous coordinates representation of $\bar{v} = (x + \Delta x, y + \Delta y)$. Thus the above matrix is also 2D Translation Matrix. Thus a 4D shear matrix on the homogeneous coordinates of a 3D vector, is effectively a 3D translation matrix. Thus any translation matrix can be represented as :

$$T\bar{v} = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{bmatrix}$$

5 Model and View Transforms

When rendering a scene in a 3D program, we need the 'World Coordinates' of an object. An object file of a model will contain all vertices in 'Object Coordinates' (i.e. w.r.t the center of the object), and the size of the object (magnitude of the vectors) would not match the desired size. The 'Model Matrix' M correctly places the object in the world.

For this we need 3 pieces of information:

1. The final size of the object. This determines the Scale applied to the object S_m.

2. The orientation of the object w.r.t the world origin. The object can be rotated any amount in the x, y, z-axes. Thus a corresponding Rotation matrix R_m is generated.

3. The position of the object in the world. Currently the center of the object is at $(0, 0, 0)$ and the object must be translated to its desired location. Thus a corresponding Translation matrix T_m is generated.

We can combine these 3 matrices in the correct order to generate the model matrix for the object. $M = T_m R_m S_m$. This matrix is calculated separately for each object in the scene and applied to all vertices of an object.

Once all the objects in a scene are placed, the camera's position must be taken into account. A vertices position w.r.t the camera is called its 'Eye Coordinates'. A 'View Matrix' does this transformation. Firstly all objects are translated by the negative of the position of the camera, using a translation matrix T_v, such that the camera is now at the origin. Next, we apply a rotation matrix R_v, that is the inverse of the camera's orientation, to all objects. Now the camera is pointing along the negative z-axis with the x and y axes being towards the right and straight up respectively.

Thus the View Matrix V can be computed as, $V = R_v T_v$. Here the translation is applied before the rotation. This View transform is applied to all vertices of all objects in the scene.

6 Projection Matrix

As a final output, we desire all vertices that fits into our field of view to be mapped to a cube from $(-1, -1, -1)$ to $(1, 1, 1)$. The x and y limits represent the bounds of the screen, and the z-axis limits represent the near

and far planes of our vision. Also if two objects are of equal size, we want the object close to the camera to occupy a larger portion of the screen. This is known as the perspective projection.

Firstly we want to map the z-range of (-near, -far) to the range $(-1, 1)$. Also we want to maintain higher numeric precision for objects close to the near plane, and less precision is sufficient for objects further away. The z-depth is used to determine which objects are in front of the other. The mapping we choose is $z' = \frac{c_1}{-z} + c_2$.

Let $n = \texttt{near}$, and $f = \texttt{far}$. Then $z' = -1$ when $z = -n$, and $z' = 1$ when $z = -f$.

Therefore $-1 = \frac{c_1}{n} + c2$ and $1 = \frac{c_1}{f} + c2$.

Solving the above equations, we get $c_1 = \frac{2nf}{n-f}$, $c_2 = \frac{f+n}{f-n}$

Also, $z' = \frac{c_1}{-z} + c_2$ can be rewritten as $z' = \frac{c_1 - c_2 z}{-z}$. The second form is useful as the numerator is a linear combination of coordinates of the vector (hence can be written as a matrix multiplication), and the division can be later achieved using homogeneous coordinates (for this we must set $w' = -z$).

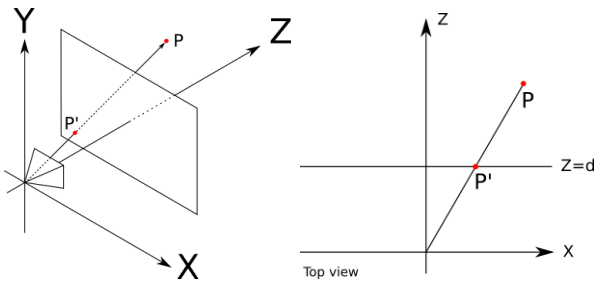

Figure 2: Perspective Projection

As seen in the above graphic, the x value should be mapped to $\frac{nx}{-z}$. Further we want the screen to only extend from $(-1, -1)$ to $(1, 1)$ thus we divide by half the screen width: $s_w = \frac{\texttt{right}-\texttt{left}}{2}$ and $x' = -\frac{nx}{zs_w}$. Similarly, half the screen height : $s_h = \frac{\texttt{top}-\texttt{bottom}}{2}$ and $y' = -\frac{ny}{zs_h}$. We can verify that the projection matrix can be then written as:

$$P\bar{v} = \begin{bmatrix} \frac{n}{s_w} & 0 & 0 & 0 \\ 0 & \frac{n}{s_h} & 0 & 0 \\ 0 & 0 & -c_2 & c_1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{nx}{s_w} \\ \frac{ny}{s_h} \\ c_1 - c_2 z \\ -z \end{bmatrix}$$

We can see that the above vector is consistent with the homogeneous

coordinates representation of $\begin{bmatrix} -\frac{nx}{s_w z} \\ -\frac{ny}{s_h z} \\ -\frac{c_1 - c_2 z}{z} \end{bmatrix}$ which is the intended vector after the perspective projection.

Thus, we can see that the final coordinates of the object can be obtained by calculating $PVM\bar{v}$. Modern graphics pipelines can perform these matrix multiplications extremely efficiently and these operations are highly parallelizable. And herein lies the benefit of homogeneous coordinates, by reducing all transformations to a matrix multiplication, we are able to leverage a single matrix multiplying unit on our hardware to compute the position data of all objects in our scene.

7 Literature References

The first reference of Homogeneous Coordinates is credited to Mobius in 1827, he defined it as a set of barycentric coordinates of a point in the interior of a triangle with respect to the vertices. It also describes how a vector in homogeneous coordinates can be scale invariant.

L.G Roberts, in 1965, implemented this idea in computer graphics and showed how homogeneous can simplify many of the affine geometric transformations we require in computer graphics.

James Blinn, in 1978, showed how homogeneous coordinates are an invaluable tool in perspective projections and how clipping vertices to the screen and maintaining a depth buffer are greatly simplified. Yet, on modern hardware, clipping line segments to the screen proves to be a bottleneck. In his 2005 paper, V. Skala, introduces a new algorithm to compute the clips of line segments using homogeneous coordinates. This algorithm can be implemented efficiently using existing hardware.

8 Conclusion

We have seen how homogeneous coordinates add a dimension to conventional Euclidean coordinates, introducing scale invariance. This property allows us to uniformly represent all relevant geometric transformations, and simplify the graphics pipeline into a series of matrix multiplication. Such a set of operations have led to the development of highly optimized and parallelized hardware in GPUs and CPUs.