The Perspective Projection Matrix

Grant Saggars

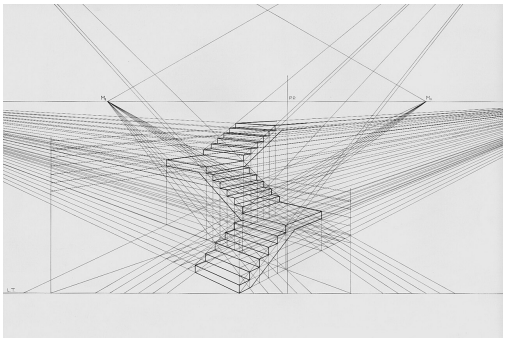
January 30, 2024

Computer graphics is a subject in which it is sometimes harder to find mathematics other than linear algebra. Of the many applications which can be found, I plan to discuss the process of perspective projection, which involves a rather interesting application of the basic principles of linear algebra that we have studied in this class.

1 Premise

Mathematically, a surface is defined as a collection of points. These collections can be defined in various ways, such as parametrically or with NURBS surfaces in CAD, or simply as vertices, edges, and faces in games, film, or art. Regardless of the definition, all of these surfaces are defined under the hood in world space coordinates. This is intuitive for accurately defining where everything in a scene is located. One might liken this coordinate system to the same used when graphing, since there is an origin at the center of the world which all axes run through, and distances between things can be measured in terms of their location relative to the origin.

In order to properly see our CAD models, games, etc., the scene must be rendered to the screen. This should immediately sound similar to a map from three-dimensional to two-dimensonal space, since our models are three-dimensional surfaces and the monitor is a two-dimensonal display. To preform this projection, we now need only derive this map.



Staircase in two-point perspective (Wikimedia)

2 Perspective

Conceptually, perspective is very simple. One could draw a ray outwards for every pixel along perspective lines (much like an artist learning to draw perspective might draw perspective guidelines). Whatever color which lies at the point hit by these lines should be the color of that pixel on the screen (where the screen is a plane we are projecting onto). Considering these vectors to be right triangles for a moment, it follows that $\frac{y'}{d} = \frac{y_1}{z_1} \rightarrow y' = \frac{y_1 d}{z_1}$. The same relation could also be derived for the x' axis, and it makes sense that z' is simply the distance d from the camera.

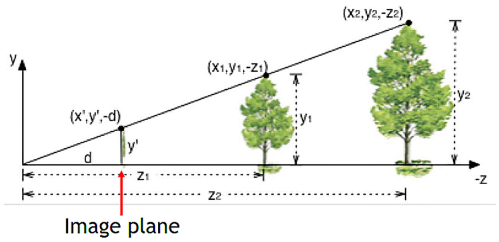


Figure 1: Diagram (UCSD)

Because the position of the projected pixel is inversely proportional to the z distance in camera space, it follows that this is a nonlinear map. To get around this, we can use a clever trick and define a new coordinate system which simply divides the x, y, z coordinates by a fourth coordinate. A vector in **A'** would therefore be: (x, y, z, w) and in **A**: $\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$. This new coordinate system is referred to as homogeneous coordinates, and it has many more applications than, and is much older than perspective projection.

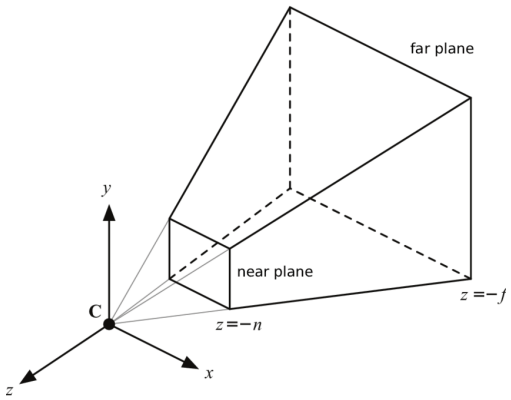
Now we can rewrite our map in homogeneous coordinates (This is only one way to write the projection matrix, and many rendering engines define the camera's z-axis to be in the negative direction, towards the camera.):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} xd/z \\ yd/z \\ d \\ 1 \end{bmatrix}$$

You might notice that this is still (practically speaking) a projection from $\mathbb{R}^3$ to $\mathbb{R}^3$. And this is actually true, despite how it might initially. The z-channel still has an important use for rendering, due to how the rendering process works. The locations of the visible pixels on the screen are mapped to the x and y coordinate, while the z coordinate is used to determine the "draw order." Our computers cannot directly tell what is in front of what in the process of projection, so under the hood objects are drawn on the screen from back to front (note that there are often more nuances in most rendering engines, especially for stylization, transparency, or optimiation reasons).

3 Clipping

It doesn't make sense to draw everything to the screen; instead, it is common to 'clip' the range of vision. This pyramid of vision is typically referred to as the view frustrum.



View Frustrum (Stefan Diewald)

We can apply a similar trick with similar triangles again to clip the near and far planes. By simimlar triangles, we derive:

$$Ps_x = \frac{(n)(P_x)}{-P_z} \tag{1}$$

where n is the distance to the near clipping plane, P_x the x-coordinate of P, and P_z the z-coordinate of P. The same process gives a similar result about the y-axis.

If we want to determine if a pixel on the screen should be visible, we can express it as:

$$l \leq Ps_x \leq r$$

where l and r are the left and right coordinates. The goal is to now normalize Ps_x so that the range of values on the x-axis in screen space is $[-1, 1]$ (the same is true for the y-axis as well). Subtracting l from the terms gives us $0 \leq Ps_x - l \leq r - l$, and dividing by $r - l$ gives $0 \leq \frac{Ps_x - l}{r-l} \leq 1$. Multiplying by 2 and subtracting 1 gives us the desired range: $-1 \leq 2\frac{Ps_x - l}{r-l} - 1 \leq 1$. Simplifying gives the normalized term:

$$\frac{2Ps_x}{r-l} - \frac{r+l}{r-l} \tag{2}$$

Encoding equation (1) and (2) in the projection matrix gives:

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The derivation for the y-coordinate is the same, and gives us the result (using t and b as additional variables):

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Now, all that is left is to derive the z-coordinate. We know that the z-coordinate can only consist of a linear combination of z and w components because of how we defined the projection, so we can solve for the z and w components. Let the z compontent equal A and the w component equal B:

$$Ps_z = \frac{0(P_x) + 0(P_y) + A(P_z) + B(P_w)}{Ps_w = -P_z}$$

Because we want to again map the z-axis to be in the range [-1,1], we can write the following system, where f is the far plane and n is the near plane:

$$\begin{cases} \dfrac{(P_z = -n)A + B}{(-P_z = -(-n) = n)} = -1 & \text{when } P_z = n \\ \dfrac{(P_z = -f)A + B}{(P_z = -(-f) = f)} = 1 & \text{when } P_z = f \end{cases}$$

$$\rightarrow \begin{cases} -nA + B = -n & (1) \\ -fA + B = f & (2) \end{cases}$$

$$\rightarrow B = -n + An \rightarrow A = -\frac{f+n}{f-n} \rightarrow B = -\frac{2fn}{f-n}$$

The final projection matrix is finally:

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

This is fundamentally the projection matrix used by most graphics libraries. If you were to choose to go build a rendering engine with OpenGL, for example, this would be the matrix used to define the camera. there are parameters which can be set to control the right, left, top, bottom, near, and far clipping planes. Often, it's easier to define these in terms of a 'field of view.' To do so:

$$\begin{aligned} \text{aspect ratio} &= \frac{\text{width}}{\text{height}} \\ \text{top} &= \tan\left(\frac{\text{FOV}}{2}\right)(\text{near}) \\ \text{bottom} &= -\text{top} \\ \text{right} &= (\text{top})(\text{aspect ratio}) \\ \text{left} &= \text{bottom} = (\text{-top})(\text{aspect ratio}) \end{aligned}$$

Note about other projections:

Perspective projection is the subject for this paper, but there also exists orthographic and isometric projection (as well as some other less common projections), which both remove the effects of perspective, keeping parallel lines parallel, and making lengths more clear. From a certain *perspective*, these other projections could also be seen as special cases or adaptations of perspective projection where the viewer could be said to be very far (infinitely) from the object.