CS 314: Doing Perspective with Matrices

Robert Bridson

September 23, 2008

1 The Perspective Z-Divide with Matrices

Last time we saw that the core of the perspective transformation is to divide camera x and y coordinates by the negative of the camera z coordinate. We then looked at extending our notion of homogeneous coordinates to include values other than 1 for w (the fourth coordinate, a.k.a. the homogeneous coordinate, in the 4D representation of 3D points). In particular, to get the 3D point a general 4D vector represents, (x, y, z, w) with $w \neq 0$, we divide through by w:

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \rightarrow \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}$$

This divide by the homogenous coordinate is called **homogenizing** the vector.

With this in place, we can implement the most basic of all perspective transformations, dividing x and y by $-z$, with the following matrix:

$$\begin{aligned} \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \\ &= \begin{pmatrix} x \\ y \\ z \\ -z \end{pmatrix} \end{aligned}$$

After this multiply, the new 4D vector can be homogenized—which is where the actual division takes place—to get the following 3D point:

$$\begin{pmatrix} x \\ y \\ z \\ -z \end{pmatrix} \rightarrow \begin{pmatrix} -x/z \\ -y/z \\ -1 \end{pmatrix}$$

You can double check that if the original w was equal to 1, as we have assumed until now, this does do exactly what we want. You can also double check that even if the original w was not equal to one—say the 4D representation of the point got multiplied by some number like 5—then our final homogenized result is exactly the same. Also note that the transformed z coordinate always ends up at -1, which destroys depth information: we'll need to fix this to preserve depth information, by including a nonzero $(3, 4)$ entry in the transformation matrix—for example:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Here the final transformed and homogenized z' becomes $-1-w/z$: although this isn't linear in the original camera space z, it at least varies monotonically with camera space z preserving depth ordering.

This construction means we can put perspective transformations in exactly the same context as all the other transformations, represented as 4×4 matrices which can be multiplied together to compose transformations, inverted, etc. The only new wrinkle is that we have to homogenize the 4D vectors to get the true 3D position in, say, normalized device coordinates just before rasterization.

Just as we made the orthogonal projection a little more sophisticated—specifying the view volume it mapped to the canonical view volume $[-1, 1]^3$, and reflecting the z coordinate to make normalized device coordinates left-handed—we can upgrade this very simple perspective transformation.

The first issue, however, is that the "view volume" of a perspective transformation is no longer a box with parallel sides: the further you get from the camera, the more is visible. This volume extends out from the camera with straight lines through the edges of the image plane, and is trimmed in camera z between a near clipping plane and a far clipping plane as before. The shape we have described is a truncated pyramid: its apex would be at the camera except for the near clipping plane which chops it off, and its base is on the far clipping plane. This shape is known as a **frustum** in general, and in this context is called the **view frustum**. The projection transformation, in the perspective case, transforms the view frustum to the canonical view volume $[-1, 1]^3$.

OpenGL, at its core, lets one define the view frustum with six parameters similar to the specification of an orthogonal transformation:

- Positive parameters `near` and `far` (n and f for short) indicate the z-extent of the view frustum: points with camera space z in the range $-f \leq z \leq -n$ are visible.
- Parameters `left` and `right` (l and r for short) indicate the x-extent of the view frustum in the near clipping plane: points with camera space x in the range $l \leq x \leq r$ and camera space $z = -n$ are visible.
- Parameters `bottom` and `top` (b and t for short) indicate the y-extent of the view frustum in the near clipping plane: points with camera space y in the range $b \leq y \leq t$ and $z = -n$ are visible.

Note that here it is crucial that `near` and `far` are strictly positive: our model of a virtual painter at the origin, painting on a sheet of glass set up on the near clipping plane, breaks down if the near clipping plane goes through the painter or is behind them. Also, as before we want the transformation to reflect z, so that normalized device coordinate z increases with distance from the camera.

The resulting matrix, as used in OpenGL for exaple (see the `glFrustum()` call), is:

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

This can be constructed from simpler transformations along with the second perspective divide matrix we saw above, if you're curious to try for yourself. A good exercise is to at least verify that this matrix maps the corners of the view frustum such as the point $(l, b, -n, 1)$ or $(rf/n, tf/n, -f, 1)$ to the appropriate corners of the canonical view volume.

1.1 Near and Far

Just as we mentioned in the orthogonal projection section, both near and far clipping planes must be specified, and should fairly closely bound the visible geometry in the scene—since eventually the transformed z coordinate may well be represented with a low precision integer, and the wider the z-range is, the worse the depth resolution gets.

The perspective transformation also has the curious effect of nonlinearly mapping z as we saw above, with a $1/z$-type relationship. This means that we get finer resolution in transformed z for objects close to the near clipping plane than with objects further away.

1.2 Aspect Ratio and Field of View

A somewhat more user-friendly way (and much more commonly used way) of specifying the view frustum is instead in terms of **aspect ratio** and **field of view**.

The aspect ratio is the width of the image divided by its height—if pixels are square, this should be the number of pixels across in the image divided by the number of pixels up and down. We haven't mentioned this before, but in retrospect it should have been clear that the orthogonal projection transformation should have also made sure that the aspect ratio (width divided by height) of the view volume should match the aspect ratio of the image, or otherwise images will appear stretched or squashed—like watching video on a wrongly configured wide screen TV.

The field of view is an angle, specifying how quickly the view frustum opens up as you go from the camera. In computer graphics (though not necessarily in other fields such as photography) this is usually defined vertically: the angle between the clipping plane on the bottom of the view frustum and the clipping plane on the top of the view frustum. To make this clear, it is often named `fovy`, standing for "field of view in y". A typical 35mm camera has a vertical field of view of about $28°$, though most interactive OpenGL programs tend to use higher angles to give a better impression of normal human vision: the human eye's vertical field of view is around $120°$, much closer to the maximum $180°$. A good starting value to try in your own programs is $90°$, which lets you see a good fraction of the scene (the smaller the field of view, the less you can see obviously), but isn't so extreme that weird stretching effects become bothersome.

In some sense, the correct field of view should be based on how tall the window on the computer screen is (measured in the real world, not in pixels!) relative to how close the eyes of the viewer are to the window. This can make the computer window seem like a real window into a virtual scene "behind the screen". However, the human eye is pretty tolerant to values quite different from this.[1] This brings up one common surprising fact that can be fairly bothersome with overly large field of view: spheres at the edges of the image get distorted into ellipses on the image. In fact, if you put your eye at the right spot (where the virtual camera is supposed to be relative to the near clipping plane) those ellipses do look like circles thanks to foreshortening, and the image looks perfectly realistic—but it may be very uncomfortable to put your eye that close to the display.

With the aspect ratio and field of view specified along with the near and far values, the usual choice (appropriate for all but some very special applications) is to make sure the view frustum is centred on the

[1] In fact, painters have often exploited the fact that humans aren't terribly picky about perspective being natural or even consistent—many classical paintings that appear extremely realistic may be seen, under close inspection, to have combined multiple inconsistent perspective projections for different parts of the scene.

camera-space z axis. This is what the popular OpenGL call `gluPerspective()` assumes.

1.3 Straight Lines and Triangles

Before we introduced the homogenization, where we divide by w, everything we had done had been a linear transformation: multiplication by a 4×4 matrix. Without homogenization, it's obvious then that lines get mapped to lines by any of the transformations, and similarly triangles get mapped to triangles. However, it's not entirely obvious the same holds true when we introduce the division.

It is intuitively obvious, just from our own real-world experience of perspective, that the 2D x and y part of a perspective-projected 3D line is a 2D line on the screen, and the same for triangles. However, it's not obvious that when you include the perspective-projected z coordinate (which varies like $1/z$) you get a straight line or a triangle in the canonical view volume. It is nonetheless possible to prove that this is indeed what happens (see Shirley's book section 7.4 for a little more detail).

However, and this fact will become important down the road when we talk about "texture mapping", barycentric coordinates in a triangle or along an edge get distorted under the perspective projection. After projection, the barycentric coordinates of a 2D point on the screen with respect to the 2D projected triangle it is in can be different from the barycentric coordinates of the point with respect to the 3D triangle in world space. This is due entirely to the division. A good mental image to keep in your mind is taking a photo of straight railway tracks receding into the distance: a point halfway along the track on the photograph (between where the track starts at the bottom of the image and where it ends at the horizon) does *not* correspond to the 3D point halfway between the camera and the horizon.