CS 314: Projection

Robert Bridson

September 18, 2008

1 Orthographic Projection

Our very primitive projection from 3D to 2D so far, namely throwing out the z coordinates once points have been put in camera space, has two severe defects. One is that it loses all information about which triangles are in front of others, so depending on the order in which triangles are rasterized, interior faces might overwrite exterior faces. The other is that triangles which are even behind the camera (have a positive camera space z coordinate) will still be drawn. This indicates that essentially we do need to keep around the z coordinate.

The first problem is ultimately resolved by **hidden surface removal** algorithms covered later in the course; we'll directly tackle the second problem now. In fact, let's do more: we have left it hazy exactly how we'll transform camera space x and y coordinates to pixel coordinates—and in particular, what the bounds of the image are in camera space (what camera x coordinate gets mapped to the left side of the image, for example).

For orthographic projection, this boils down to a 3D version of the 2D windowing transformation we saw before (that used a combination of translation and scaling to map between two rectangles). In particular, we will let the user specify a **view volume** in camera space, an axis-aligned box identifying all the visible parts of the scene:

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

twist: points with camera space z in the range $-f \leq z \leq -n$ are visible.

Note that `near` and `far` get negated when describing the z-extent: the right-handed convention means that points with negative camera space z are in front of the camera, but most people prefer to think of z (when at the projection stage) as a notion of distance from the camera. The parameter `near` should be positive, indicating the nearest that points can be to the camera and still be visible, and `far` should be strictly greater, indicating the furthest visible points can be.

The **projection transform** maps this view volume $[r, l] \times [b, t] \times [-n, -f]$ to the system's **canonical view volume**, which by convention is the cube $[-1, 1] \times [-1, 1] \times [-1, 1]$ centred on the origin. The result of applying the projection transform to camera space coordinates are **normalized device coordinates**. This new coordinate system is left-handed: its x-axis points right and its y-axis point up like in camera space, but its z-axis points in towards the screen instead of out, so that $z = -1$ is the near side of the volume and $z = 1$ is the far side. After projection, smaller z values indicate closer objects than bigger z values. In this convention, we often call the z coordinate the **depth** of the point.

Including this reflection of z, it's not too hard to work out the orthographic projection. In 4×4 matrix form it is:

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

Note the $(3, 3)$ entry is negative, consistent with a reflection in z. You could also build this, as before in 2D, as a sequence of translation, scaling (with a negative z scale factor for reflection) and another translation.

Sometimes the parameters describing the user's view volume are called **clipping planes**. Much like we clipped triangles to the bounds of the image when rasterizing them, in 3D we clip geometry to the bounds of the view volume before proceeding further down the pipeline (see the next section). In particular, we ignore or **cull** geometry that lies on the wrong side of the near clipping plane, since we can think of it as being behind the camera. However, you might be wondering if it's really necessary to have to specify a far clipping plane as well, culling geometry that's on the right side of the camera but too far away. In fact, usually the far clipping plane is chosen to be far enough that all geometry in the scene can be included. However, it's critical not to choose it too large since eventually the projected z values get rounded and stored; particularly on lower end graphics cards only an 8-bit integer might be used, allowing for serious errors if those 256 possible discrete depth values have to cover a needlessly large z range.

2 The Pipeline

With this in place, we can put a little more detail into our notion of a practical 3D rendering pipeline:

- Input 3D geometry in world space coordinates is transformed by the model-view matrix to...
- camera space coordinates, which are transformed in turn by the projection matrix to...
- normalized device coodinates. Geometry outside the canonical view volume is culled, and the rest gets transformed to...
- output coordinates: the x and y coordinates can then be transformed in 2D to pixels, and typically the depth z gets transformed to the smaller range $[0, 1]$ for use in hidden surface removal. The triangles are rasterized, producing...
- the final image

3 Perspective Projection

Orthographic projection, seen above, is useful in some engineering or scientific contexts, since it has the special property that an object projects to the same size on screen no matter how far away it is. However, it's not how we see the world: we see perspective **foreshortening**, where further objects appear smaller than nearer objects.

3.1 Introducing the Z-Divide

As we mentioned before, a good model for rendering is to think in terms of simulating the light recorded by a virtual camera in a virtual world. The actual formation of an image in a camera is rather complicated however (involving lenses, apertures, and flipping the image upside-down for starters) so we'll adopt an even simpler model now. Imagine a virtual painter, with one eye located at the origin in camera space (and the other eye closed), and a sheet of glass set up in front of them along the near clipping plane in camera space. The painter looks at the scene through this sheet of glass, and whatever they see they paint directly on the glass. The painting on the glass is the correct perspective projection of the scene—that 2D painting is what gets displayed on the 2D screen. We did this in class, looking through the windows and outlining whatever we saw with dry-erase markers; you might want to try it at home to make this more concrete in your understanding.

With this model in place, we can then start solving the mathematical problem of describing the projection. Let's say there is a point (x, y, z) in camera space coordinates, the near clipping plane is at $z = -n$, and we need to determine the coordinates $(x', y', -n)$ of the point's perspective projection (according to the virtual painter's eye) on that clipping plane. Note that the straight ray of light coming from (x, y, z) to the painter's eye at the origin $(0, 0, 0)$ has to pass through the projected point $(x', y', -n)$. Therefore x' and y' are solutions to the problem of finding the line that goes through $(0, 0, 0)$ and (x, y, z) and evaluating it at $z = -n$; you can also tackle this using similar triangles as in class. Either way, the answer is to scale x and y by $-n/z$:

$$\begin{pmatrix} x' \\ y' \\ -n \end{pmatrix} = \frac{-n}{z} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -n\frac{x}{z} \\ -n\frac{y}{z} \\ -n \end{pmatrix}$$

The crucial aspect of this, what gives us perspective, is that the x and y coordinates are divided by z to get the projected x' and y'.

Unfortunately, as much as we love matrices, there is no way to represent division using a matrix. However, there's also no obvious way to represent vector addition with matrix multiplication, and we got around that by going to homogoneous coordinates when we covered translation. One of the most lovely tricks in computer graphics is the fact that homogeneous coordinates can come to the rescue here too!

3.2 Homogeneous Coordinates

So far, we have assumed the fourth "homogenous" coordinate of a 3D point is always equal to one. We'll now relax that, and allow a point to have any nonzero value for its homogenous coordinate; we'll call it w in general. We'll also introduce an equivalence relation, a rule declaring that two 4D vectors can represent the same 3D point:

> In homogenous coordinates, two 4D vectors (x_0, y_0, z_0, w_0) and (x_1, y_1, z_1, w_1) represent the same 3D point if and only if $x_0/w_0 = x_1/w_1$, $y_0/w_0 = y_1/w_1$, and $z_0/w_0 = z_1/w_1$.

In particular, we can associate any 4D vector (x, y, z, w), where $w \neq 0$, with the 3D point $(x/w, y/w, z/w)$, or alternatively pick $(x/w, y/w, z/w, 1)$ as the representative of the equivalence class it belongs to. You can double check this notion of equivalence is perfectly compatible with all the transformation matrices we've seen so far (in particular, translation, which was the only one which really needed the homogeneous coordinate).

So now, if multiplication by a 4×4 matrix changes the homogenous coordinate w to something other than 1, the resulting point implicitly needs a division to figure out where in regular geometric 3D space it is. So far all the transformation matrices we have seen had their bottom row equal to $(0\ 0\ 0\ 1)$, indicating the homogeneous coordinate is unchanged; you probably can guess right away how we need to change that bottom row to get the effect of a divide-by-z for perspective.