Jim Blinn's Corner

You can use homogeneous coordinates to interpolate various parameters properly when tiling polygons. This technique is based completely on one of those things that homogeneous coordinates are good at—perspective.

Hyperbolic Interpolation

James F. Blinn
California Institute of Technology

It's always a red letter day when I can figure out a new use for homogeneous coordinates. This time I'll tell you about a way to use them to interpolate various parameters properly when tiling polygons. The exact definition of "properly" comes from one of those things that homogeneous coordinates are good at—perspective.

The existing machinery

First, some notational conventions: I'll write matrices in boldface, vectors in Roman type, and vector elements in italics with subscripts. A general homogeneous vector, one whose w component is not 1, will appear with a tilde over the name. A vector of the same name with no tilde represents that homogeneous vector with the w component divided out.

Coordinate systems

Now let's review some basic operations of the graphics pipeline and define some coordinate systems. In general, the pipeline transforms a coordinate point through a whole chain of coordinate spaces as the point makes its way to the screen. I'm going to vastly simplify the process for this discussion. There are only two coordinate spaces that we'll really need to deal with here: eye space and pixel space.

We'll start with polygon vertices in *eye space*, the coordinate space with the eye at the origin looking down the z axis. This space is significant because it's the last step in the chain in which physical distances are meaningful. For example, it's where we must perform all lighting calculations. Let's call a point in this space E. In homogeneous coordinates this is

$$\mathrm{E} = [E_x, E_y, E_z, 1]$$

We do the perspective distortion necessary to get to hardware pixel space in two steps. First, we multiply by a 4×4 matrix consisting of a perspective transformation and a viewport transformation. I'll call this matrix **M**. Since **M** has a perspective component, the w coordinate of the transformed point will not be 1.

$$\mathrm{E}\mathbf{M} = \tilde{\mathrm{P}} = [\tilde{P}_x, \tilde{P}_y, \tilde{P}_z, \tilde{P}_w]$$

We perform clipping in this coordinate system. (This is different from the optimized clipping space I discussed in "A Trip down the Graphics Pipeline: Line Clipping," *IEEE CG&A*, Vol. 11, No. 1, Jan. 1991, pp. 98-105. The method I described there merely changes the coefficients of the clipping planes used; it doesn't change the basic mathematical relationships).

After clipping, we perform the second operation; we divide out the w component to get coordinates in nonhomogeneous hardware pixel coordinates.

$$\tilde{\mathrm{P}} / \tilde{P}_w = \mathrm{P} = [P_x, P_y, P_z, 1]$$

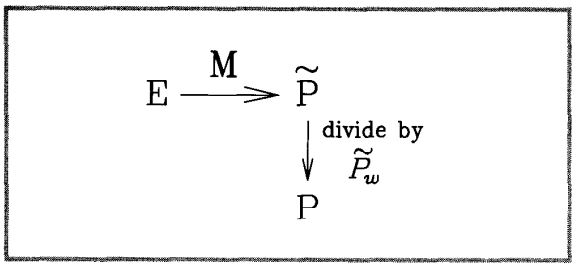

Figure 1. Transformations between coordinate systems.

I diagrammed these relations in Figure 1. The key thing that we are going to worry about here is manipulating coordinates either *before* or *after* the homogeneous division.

Polygon tilers

Now let's quickly review the mathematics of a polygon tiler. A 3D polygon tiler begins with a list of the pixel space coordinates of each vertex, $[P_x, P_y, P_z]$. The tiler must then do two things. First, it must identify which pixels lie inside the polygon. Second, it must calculate a Z depth value for each pixel inside the polygon for use in occlusion tests. We are primarily interested in this second calculation.

A tiling algorithm consists of two nested loops, representing two successive reductions in dimensionality. The outer Y loop tracks the intersection of each edge with a *current scan line*. That is, the Y loop interpolates values for $[P_x, P_z]$ between the endpoints of the edge as a function of the pixel coordinate Y. The inner X loop then interpolates the $[P_z]$ value horizontally between pairs of edge intersections.

Let's examine the mathematics of the Y loop more closely. We are given coordinates at two endpoints of an edge; let's call them P′ and P″. The edge intersects scan line Y at a proportional distance between these two points of

$$\alpha = \frac{Y - P'_y}{P''_y - P'_y}$$

α goes from 0 at P′ to 1 at P″. The intersection is then

$$P = P' + \alpha(P'' - P')$$

We step down the screen in equal steps of Y. This means we are going to evaluate the above equation for equal steps in α. We typically do this incrementally; we precalculate the change in P resulting from each scan-line jump and then, for each iteration of the Y loop, add that increment to P.

There is an implicit assumption here. The Y, X nested interpolation works consistently only if the polygon is *well behaved*. By that I mean that all the vertices in P space are coplanar. Effectively, the relation of P_z to pixel coordinates is of the form

$$P_z = aX + bY + c$$

Another question

But there's another question about why this works. What we actually start with is a flat polygon in *eye* space. We then put this polygon through a weird perspective-distorting transform and expect that mere linear interpolation in pixel space gives us the correct Z value. To show that this is really okay, let's start with a point that is a distance β along an edge in eye space.

$$E = E' + \beta(E'' - E')$$

Transform it by **M** to get $\tilde{P}$

$$\tilde{P} = E'\mathbf{M} + \beta(E''\mathbf{M} - E'\mathbf{M}) = \tilde{P}' + \beta(\tilde{P}'' - \tilde{P}')$$

Divide out the w component to get P.

$$P = \frac{\tilde{P}' + \beta(\tilde{P}'' - \tilde{P}')}{\tilde{P}'_w + \beta(\tilde{P}''_w - \tilde{P}'_w)}$$

Now we want to write this in terms of the endpoints of the edge in pixel space. Each endpoint satisfies

$$P' = \frac{\tilde{P}'}{\tilde{P}'_w}$$

or

$$\tilde{P}' = P'\tilde{P}'_w$$

Plug this and a similar expression for $\tilde{P}''$ into the above equation, do a little algebra, and you get

$$P = P' + \left(\frac{\beta\tilde{P}''_w}{\tilde{P}'_w + \beta(\tilde{P}''_w - \tilde{P}'_w)}\right)(P'' - P')$$

What does this mean? Compare this with linear interpolation in pixel space:

$$P = P' + \alpha(P'' - P')$$

It means that a point, say halfway between E′ and E″, transforms to a point *somewhere* on the line connecting P′ and P″, but *not* necessarily halfway between them. To generalize, equally spaced dots along an edge in eye space transform into dots that are indeed colinear in pixel space; they just aren't equally spaced any more. This means that a flat polygon in eye space transforms into a flat polygon in pixel space.

Getting colorful

With the invention of Gouraud shading, polygon tilers began linearly interpolating colors across the polygon in the

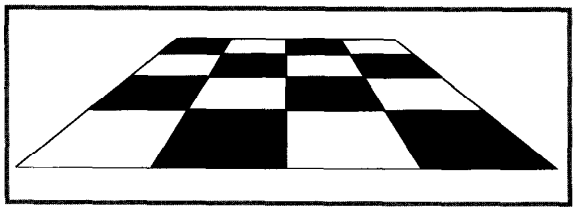

Figure 2a. Correct perspective.

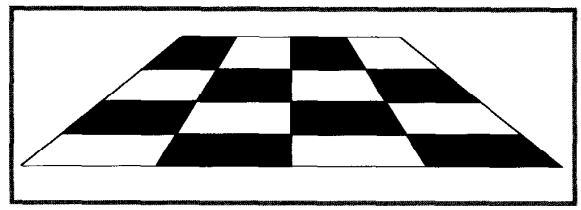

Figure 2b. Incorrect perspective.

same way they interpolated Z values. To make the tiler do this, modify the machinery as follows. For each vertex, build a larger vector of values that include colors as well as position coordinates. Each vertex looks like

$$[E_x, E_y, E_z, 1, C_{\text{red}}, C_{\text{green}}, C_{\text{blue}}]$$

Then feed this down the pipe. The first stage transforms the positional components according to matrix **M**:

$$[\tilde{P}_x, \tilde{P}_y, \tilde{P}_z, \tilde{P}_w, C_{\text{red}}, C_{\text{green}}, C_{\text{blue}}]$$

This then goes to the clipper. Any interpolation done here is performed on the color components as well as the positional components. (The perceptive reader might suspect that there is something fishy about the way we are clipping colors. We will deal with this later.)

After clipping, divide the w component out of the positional components

$$\left[\frac{\tilde{P}_x}{\tilde{P}_w}, \frac{\tilde{P}_y}{\tilde{P}_w}, \frac{\tilde{P}_z}{\tilde{P}_w}, C_{\text{red}}, C_{\text{green}}, C_{\text{blue}}\right]$$
$$= [P_x, P_y, P_z, C_{\text{red}}, C_{\text{green}}, C_{\text{blue}}]$$

The tiler gets an array of this form for each vertex. The tiler then uses the same machinery to interpolate color values as it uses to interpolate P_z values.

To get consistent results regardless of how the polygon might be rotated, we must have well behaved color assignments. I use *well behaved* here in the same sense I did for P_z values. For, say, the red color primary, the vertex color assignments expressed as $[P_x, P_y, C_{\text{red}}]$ should be coplanar (with a similar requirement for green and blue values). This makes the color a linear function of the screen space coordinates; that is, each color primary is expressible in the form

$$C_{\text{red}} = aX + bY + c$$

Of course this is guaranteed if the polygon is a triangle, but it's perfectly possible for polygons with more sides. If the color assignments don't satisfy this constraint, then—according to most computer graphics books—you split the polygon into triangles. But, as I explain below, this isn't an adequate solution.

More stuff

Why stop at color? Phong shading requires that we interpolate normal vector components between endpoints. No problem. Just treat each component the same way we treated color components.

How about some more elaborate shading models? Suppose that we have local light sources and that our lighting model depends on the location of the viewer. For each point on the object, a different vector extends from it to the light source and to the eye. We therefore need to calculate the eye space point E that corresponds to each pixel inside the polygon.

And then there's texture mapping. We can assign texture coordinates (u, v) to each vertex and interpolate them at each pixel, then use the interpolated values as input to some texture function.

The naive approach is to just tack on more elements to our array

$$[P_x, P_y, P_z, C_{\text{red}}, \ldots, N_x, \ldots, E_x, \ldots, u, v]$$

and linearly interpolate them all between polygon vertices. But there's a hidden error here. Doing pure linear interpolation in screen space for E, u, and v is really not correct. To appreciate this fully, let's look at a common example.

A problem of perspective

Suppose we have a square polygon with parametric coordinates (u, v) defined at the four corners as (0, 0), (0, 1), (1, 0), and (1, 1). Now let's be really original and map a checkerboard onto the square and view it in perspective. If we simply interpolate the (u, v) parameters linearly across and down the polygon according to the standard tiling machinery, we get the checkerboard you see in Figure 2b, with equal vertical spacing of the small squares. This is not right. How can you tell? Use a standard artist's trick: draw a diagonal through the perspectivized square. If the perspective is correct, the corners of the small squares should pass through this line. Try it; I'll wait. ... Not too good, right? What we really want is for the checkerboard to look like Figure 2a. Try drawing the diagonal line now, and you'll see that it works.

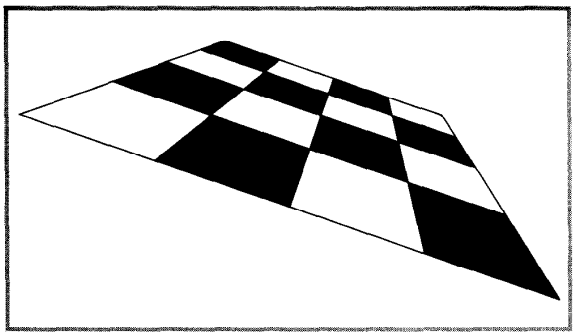

Figure 3a. Correct perspective.

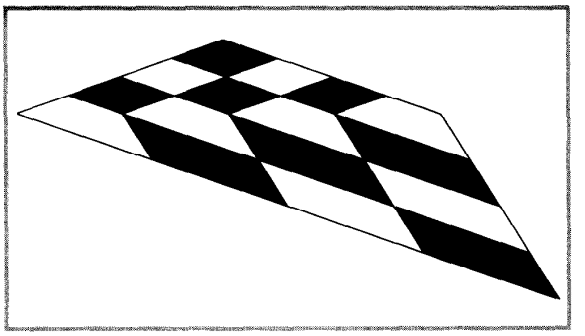

Figure 3b. Incorrect perspective.

But this isn't the worst thing that can happen. Try rotating the square a bit. You'd like to see the nice picture in Figure 3a. Instead, you'll get the weird mess shown in Figure 3b. Let's see how this comes about. Figure 4a shows the results of v interpolation. As the tiler scans out the top half of the polygon, v stays a constant 0 on the left edge and interpolates from 0 to 1 on the right edge. Interpolating across each scan line then gives the constant v lines shown. On the bottom part of the polygon, v interpolates from 0 to 1 on the left edge and stays a constant 1 on the right. Interpolating across each scan line gives the constant v lines shown. They are all unpleasantly bent. It's not quite so bad for interpolation of u values shown in Figure 4b. Here the lines aren't bent, but they still are incorrectly equally spaced.

Correct mapping

Let's figure out how to do this correctly. This turns out to be easier if we change the problem slightly. Let's instead solve the problem of finding eye space coordinates E at each pixel. First we'll do this simply but stupidly. We simply take each interpolated pixel space coordinate $[P_x, P_y, P_z]$ and transform it by the inverse of **M** to get back to eye space. This will produce something with a nonunit w component, so we have to divide it out to get back to true eye space.

$$P\mathbf{M}^{-1} = \tilde{E} = [\tilde{E}_x, \tilde{E}_y, \tilde{E}_z, \tilde{E}_w]$$
$$\tilde{E} / \tilde{E}_w = E$$

A faster way to be correct

Doing a full matrix multiplication at each pixel is slow and, fortunately, unnecessary. Look at the above equation again.

$$\tilde{E}=P\mathbf{M}^{-1}$$

Remember that we are calculating the screen space vector P by linearly interpolating between endpoints in pixel space. Since P is related to $\tilde{E}$ by a nice linear matrix multiplication, we can effectively "factor" the matrix multiplication out of the loop by linearly interpolating between $\tilde{E}$ values at its endpoints. We only need to find the $\tilde{E}$ coordinates at each endpoint of the edge.

In general, for any point $\tilde{E}$

$$\tilde{E} = P\mathbf{M}^{-1} = \frac{\tilde{P}}{\tilde{P}_w}\mathbf{M}^{-1} = \frac{\tilde{P}\mathbf{M}}{\tilde{P}_w}\mathbf{M}^{-1}$$

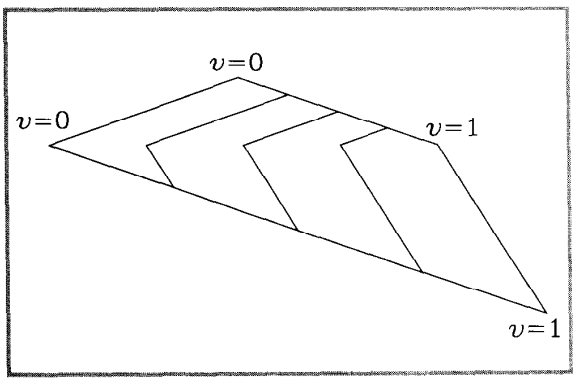

Figure 4a. Lines of constant *v*.

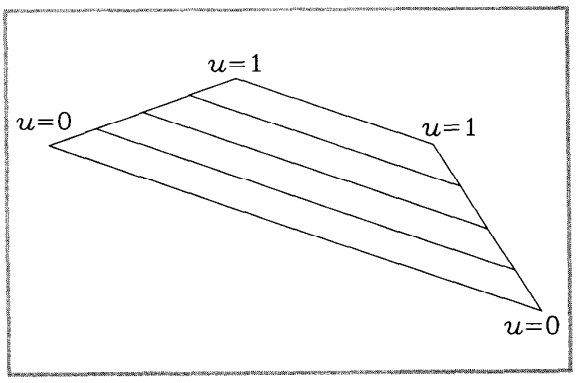

Figure 4b. Lines of constant *u*.

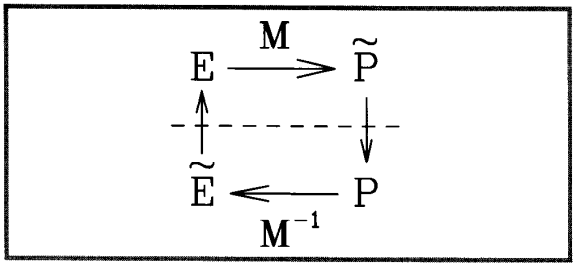

Figure 5. Division between eye space and pixel space and their linearly related spaces.

so

$$\tilde{\mathrm{E}} = \frac{\mathrm{E}}{\tilde{P}_w}$$

We can see this in Figure 5, an enhanced version of Figure 1. Crossing over the dotted line represents a homogeneous division.

What does this mean? To feed our tiler, we manufacture an array of values for each polygon vertex as follows:

$$\left[P_x, P_y, P_z, \frac{E_x}{\tilde{P}_w}, \frac{E_y}{\tilde{P}_w}, \frac{E_z}{\tilde{P}_w}, \frac{1}{\tilde{P}_w}\right]$$

$$= \left[P_x, P_y, P_z, \tilde{E}_x, \tilde{E}_y, \tilde{E}_z, \tilde{E}_w\right]$$

The first three elements are the positional coordinates in postperspective pixel space. The last four are values which, when linearly interpolated along edges, give four homogeneous coordinates of the point in eye space. You can use the same interpolation technique used to interpolate the P_z value. We still must divide, on a pixel by pixel basis, the interpolated $\tilde{E}_x$, $\tilde{E}_y$, and $\tilde{E}_z$ values by the interpolated $\tilde{E}_w$ value to get the true eye-space vector E.

We have basically shown that while we can interpolate P_z linearly since it's of the form

$$P_z = aX + bY + c$$

we should interpolate, say, E_z *hyperbolically* since it is of the form

$$E_z = \frac{aX + bY + c}{dX + eY + f}$$

This hyperbolic interpolation is just the quotient of two linearly interpolated quantities. We might have seen this by the relation between α and β under the "Another question" section earlier.

Again, referring to Figure 5, we find that linear interpolation of coordinates below the dotted line implies hyperbolic interpolation for coordinates above the line. And vice versa. Note, however, that while each *component* of E is a hyperbolic function of P, all the points of E are still coplanar.

Now back to texture parameters. To consider them well behaved, we expect that they are related to eye space coordinates by another nice linear function. We can then calculate them with the same machinery we use to calculate the eye space coordinates. We build a vector of

$$\left[P_x, P_y, P_z, \frac{u}{\tilde{P}_w}, \frac{v}{\tilde{P}_w}, \frac{1}{\tilde{P}_w}\right]$$

Interpolate the last three values across the polygon just as before. Then at each pixel divide the interpolated $u / \tilde{P}_w$ and $u / \tilde{P}_w$ by the interpolated $1 / \tilde{P}_w$. It works.

Clipping

Whenever you see a division in an expression, you should immediately be worried by the possibility that the denominator might be zero. What does this mean here? If $\tilde{P}_w = 0$, it means that the point P is at infinity. This will come from a point in eye space that's in the same plane as the eye, that is $E_z = 0$. This is a perfectly reasonable situation, and we have to be able to deal with it.

Why didn't we worry about this when we did the homogeneous division of $\tilde{\mathrm{P}}$ by $\tilde{P}_w$ to get P? Because these parts of the polygon are removed by the clipping process. Revelation! We want to defer the division by $\tilde{P}_w$ of all our other auxiliary coordinates until *after* clipping. Now all we have to show is that this generates the correct values geometrically.

Clipping happens in postperspective space before the homogeneous division. That is, it happens to $\tilde{\mathrm{P}}$ coordinates and is itself a linear interpolation of the points in homogeneous space. Let's say our edge from $\tilde{\mathrm{P}}'$ to $\tilde{\mathrm{P}}''$ straddles a clip boundary. The clipper calculates a proportional distance γ where the edge hits the boundary. The clipped point is then

$$\tilde{\mathrm{P}}''' = \tilde{\mathrm{P}}' + \gamma\,(\tilde{\mathrm{P}}'' - \tilde{\mathrm{P}}')$$

We then divide this interpolated point by its interpolated w component

$$\tilde{P}'''_w = \tilde{P}'_w + \gamma(\tilde{P}''_w - \tilde{P}'_w)$$

to give the endpoint of the clipped edge in pixel space

$$\frac{\tilde{\mathrm{P}}'''}{\tilde{P}'''_w} = \mathrm{P}''' = [P'''_x, P'''_y, P'''_z, 1]$$

Now to apply this to $\tilde{\mathrm{E}}$ interpolation. The $\tilde{\mathrm{E}}$ vector corresponding to this clipped P is

$$\tilde{\mathrm{E}}''' = \mathrm{P}'''\mathbf{M}^{-1} = \frac{\tilde{\mathrm{P}}'''\mathbf{M}^{-1}}{\tilde{P}'''_w} = \frac{(\tilde{\mathrm{P}}' + \gamma\,(\tilde{\mathrm{P}}'' - \tilde{\mathrm{P}}'))\mathbf{M}^{-1}}{\tilde{P}'_w + \gamma\,(\tilde{P}''_w - \tilde{P}'_w)}$$

$$= \frac{\mathrm{E}' + \gamma\,(\mathrm{E}'' - \mathrm{E}')}{\tilde{P}'_w + \gamma\,(\tilde{P}''_w - \tilde{P}'_w)}$$

That is, if we clip the E vector just as we do the $\tilde{\mathrm{P}}$ vector and *then* divide it by the clipped $\tilde{P}_w$ value, we will get the correct $\tilde{\mathrm{E}}$ vectors to interpolate between in the polygon tiler. We are guaranteed that $\tilde{P}_w \neq 0$ because the clipper is designed to clip

away just those types of points. Notice that clipping applies to coordinates above the dotted line of Figure 5. The moral: Clip first and divide later.

More about color

So what about colors C and normal vector components N? So far we have been interpolating them linearly in P space. This sounds okay, but consider the following. Suppose you had a square polygon and your shading calculations gave one color to the two left vertices and another to the two right vertices, a simple gradation of color across the square. You might think that this would be well behaved according to our earlier definition. But it's not well behaved if the square is viewed in perspective. Look again at Figure 4a and pretend that v represents color. Ick. And dividing it into triangles definitely won't help, since that is effectively what the tiler did for the orientation I picked for Figure 4a.

Now, admittedly, real shading situations aren't so violent, because the values of the colors and normal components aren't usually radically different for the various vertices. A lot of rendering programs don't worry about it for this reason. But it's not too hard to do it right; we interpolate colors hyperbolically. And when you think about it, it really makes the most sense to define color values and normal vector components so that they are linearly interpolated in preperspective eye space, just like texture coordinates. This is a sensible approach because distance measurements still make sense in eye space.

The added bonus is that interpolating colors hyperbolically means that we clip the colors correctly.

The new mechanism

So here's the whole story.

1. Construct an array of values for each vertex of the polygon

$$[\tilde{P}_x, \tilde{P}_y, \tilde{P}_z, \tilde{P}_w, C_{red}, E_x, \ldots, N_x, \ldots, u, \ldots, 1]$$

The auxiliary components following $\tilde{P}_w$ can be any values you intend to linearly interpolate in nonperspective-distorted eye space. Note the constant 1 at the end.

2. Perform the standard clipping process, interpolating *all* values if any clipping is done.

3. After clipping, it's time for homogeneous division. In the original algorithm, we just divided $\tilde{P}$ by $\tilde{P}_w$. Now, we divide the *entire* array by the $\tilde{P}_w$ value—colors and all. This gives

$$\left[\frac{\tilde{P}_x}{\tilde{P}_w}, \frac{\tilde{P}_y}{\tilde{P}_w}, \frac{\tilde{P}_z}{\tilde{P}_w}, 1, \frac{C_{red}}{\tilde{P}_w}, \ldots, \frac{E_x}{\tilde{P}_w}, \ldots, \frac{N_x}{\tilde{P}_w}, \ldots, \frac{u}{\tilde{P}_w}, \ldots, \frac{1}{\tilde{P}_w}\right]$$

or

$$[P_x, P_y, P_z, 1, \tilde{C}_{red}, \ldots, \tilde{E}_x, \ldots, \tilde{N}_x, \ldots, \tilde{u}, \ldots, \tilde{1}]$$

This final value, which I have fancifully written as $\tilde{1}$, is the homogeneous coordinate—the w value—for each of the auxiliary vectors we are interpolating. That is

$$\tilde{1} = \tilde{C}_w = \tilde{E}_w = \tilde{N}_w = \tilde{u}_w$$

4. Linearly interpolate all values down polygon edges and across scan lines inside the polygon. Remember, the value of $\tilde{1}$ is different at the various vertices, so it too changes across the polygon.

5. At each pixel, divide the auxiliary components ($\tilde{C}$, $\tilde{E}$, $\tilde{N}$, and $\tilde{u}\tilde{v}$) by the final element $\tilde{1}$ to get the proper perspectively projected value. We are guaranteed that $\tilde{1}$ will never be zero. Why? Because, after clipping, all $\tilde{P}_w$ values are positive.

6. Calculate the pixel color using these values as input to some shading model.

There is another organizational thing going on here. The clipping and homogeneous division, even the tiler, do not need to know the meaning of the auxiliary components. They can operate just fine given only the total length of the array. The only code that needs to know the array's interpretation is the code that feeds vertices into the pipe and the code that colors the pixels that come out of it—that is, Steps 1 and 6. This means that you can make a system that lets users select which and how many things to interpolate without needing to change Steps 2 through 5.

Anything else?

This new way to do interpolation is pretty simple. Within the polygon tiler, we have just one more extra value, $\tilde{1}$, to interpolate. But then we must divide this into all our auxiliary variables on a per pixel basis. Can we get rid of the nasty old division?

If the $\tilde{1}$ values on each endpoint are equal, you can divide it out before interpolating. This happens if the polygon is parallel to the screen—and this is perhaps not too likely.

We could approximate the hyperbolic curve we want by some sort of higher order polynomial approximation. But that's probably not worth it. It's best to be correct and just do the division. It'll make you feel warm inside. ❑