Texture Mapping Part IV: Approximations

Chris Hecker

So you thought we were done with perspective texture mapping. Didn't you feel something was missing? The penultimate article in this series helps shed some light on the ins and outs of approximations.

Knowing, understanding, and following through on your goals are key parts of software development. Often, I'll go into a project with a perfectly valid set of goals, only to get distracted along the way and produce something that meets an entirely different set of goals, but completely misses my original ones. This phenomenon seems to be pretty common in the game industry as well, where companies go into a project with the goal of creating a great game, but end up creating a nice piece of technology with no game play.

Similarly, our goal during this series is *not* to produce a perspective texture mapper. Surprise! Our goal is actually to draw perspective-texture-mapped triangles on the screen quickly. A subtle but important difference exists between these two goals. As with most things in real-time PC graphics, the result on the screen is the only thing that matters, not how you got it there. We can exploit the difference between what *looks* right and what *is* right and get big speedups in our code.

In other words, if a beautifully written, mathematically perfect texture mapper and a total hacked piece of junk produce the exact same results on the screen (including avoiding jitter and all the other things we've been learning), and the hack is 10 times faster, then the choice is clear if you're interested in speed. It's important to note that this does not mean we've been wasting our time learning about "correct" perspective texture mapping. In fact, it's just the opposite. Now that we intimately understand how the math works, we're in a much better position to throw it all out and cut corners.

In Our Last Episode...

Let's quickly summarize and tie up loose ends from my last column in this series, "Perspective Texture Mapping, Part III: Endpoints and Mapping" (Behind the Screen, Aug./Sept. 1995). The summary is pretty short: we've developed a complete, high-quality sub-pixel-accurate perspective texture mapper.

The only loose end we've got left (besides performance, which is the main subject of this article) concerns the real-to-integer texture coordinate mapping. When we left off, we had a bug in this mapping and we needed to choose a rounding rule to get the correct mapping. I hinted that we already had the information available to make the decision on which rounding rule to use, but I didn't give the answer. As many of you probably guessed, the gradients are the key to making this decision (which implies you must switch between rounding rules at runtime—and this is indeed the case). Unfortunately, limited space keeps me from going into the derivation of the solution. If you're interested, you can pick up the sample code I mention at the end of this column on the *Game Developer* ftp site. You'll find a big comment block explaining things there.

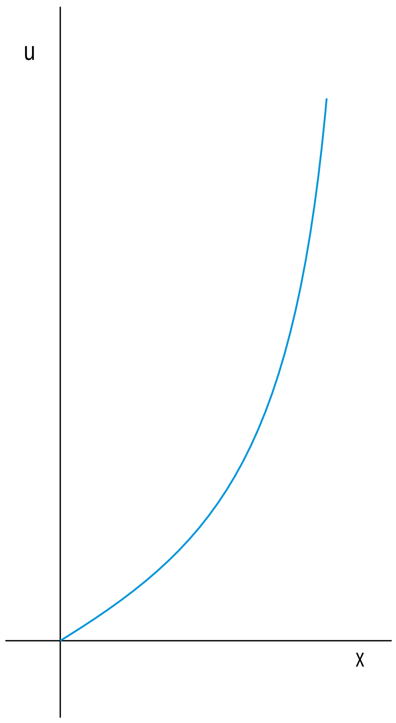
Divided We Fall

Finally I'm ready to make good on the second half of my two-part promise: the first part of which was to develop an easy-to-understand perspective correct texture mapper, and the second to speed it up to interactive performance.

In our efforts to speed up the mapper the obvious question is, "Where is the code currently spending its time?" I ran a profiler on it and ended up with what I like to call the perfect profile—almost all our time (96%) is spent in one function, `DrawScanLine`. The nice thing about a profile like this is that your optimization work in that function, sometimes called the hotspot, is very highly leveraged. In other words, every little bit you speed up the hotspot increases the overall performance by a lot. It's not surprising that `DrawScanLine` is the culprit because it contains the pixel loop, but it's always good to check our assumptions and gather some real data.

Listing 1 gives us a closer look at `DrawScanLine` inner loop. In it, we see that the function is doing a divide and two multiplies per pixel to figure out the texture coordinates, a multiply to calculate the texture offset, and a few adds. The divide is probably the major sink here. Divides are much slower than multiplies on most processors, and multiplies are generally slower than adds. If we comment out the divide per pixel and do a linear interpolation between the left and right edge, the mapper performs seven times faster in my test (and of course looks totally wrong because there's no perspective correction). Obviously, getting rid of the divide helps a lot. However, to get rid of the divide and keep the same visual quality we need to know exactly what the divide is doing there in the first place.

Figure 1. The Perspective Curve



Listing 1. The Inner Loop

```
while(Width-- > 0) {
    float Z = 1/OneOverZ;
    int U = UOverZ * Z + 0.5;
    int V = VOverZ * Z + 0.5;

    *(pDestBits++) = *(pTextureBits + U + (V * TextureDeltaScan));

    OneOverZ += Gradients.dOneOverZdX;
    UOverZ += Gradients.dUOverZdX;
    VOverZ += Gradients.dVOverZdX;
}
```

The divide is the crux of the perspective texture mapper. It takes the linear interpolations of 1/z, u/z, and v/z and turns them into the nonlinear curve that samples pixels close together when the polygon is near the eyepoint and samples farther apart when the polygon is distant. If the polygon is slanted so that it's close on one end and distant on the other, the divide smoothly moves from close to separated samples. Figure 1 shows a plot of screen x versus sampled u for a typical perspective mapped scanline. As x increases, u starts increasing slowly and then grows much quicker. Obviously, this data is from a polygon that's close at low values of x and distant at larger values.

This curve (it's different for each polygon and scanline, in general) is what makes perspective mapping look correct, so the trick is to approximate the curve without using a divide and without adversely affecting our visual quality.

The Big Three

When I say we want to approximate the perspective curve, I mean we want to use alternate equations that will produce the same—or very close to the same—output (the u value shown in Figure 1) for the same input (x, as shown in Figure 1), but hopefully will be more efficient than our algorithm with its divides.

Three approximations to the perspective texture mapping equations are commonly used: subdividing affine, quadratic, and lines-of-constant-z.

I'm going to talk about lines-of-constant-z first because it's very different from the other two. First, "lines-of-constant-z" is an awkward name. Some people choose to call it "free-direction texture mapping," which is a bit easier to say but not quite as precise. Basically, a lines-of-constant-z rasterizer tries to take advantage of the neat fact that there are straight and parallel lines that have a constant z through any planar polygon. Imagine you're sliding a plane that's perpendicular to the z-axis back through your polygon. The plane will slice the polygon in a series of parallel lines as it moves through the depth range occupied by the polygon. All the pixels along one of these lines will have the z value of the plane, so the line has a "constant z."

Now, if you look at the math behind our projection, when z is constant, our equation turns into a linear interpolation (the perspective curve from one point on this line to the next is a line itself), which is quick and easy to compute and has no divides in the pixel loop. The down side is that in general you'll be interpolating along a diagonal line in the destination instead of across a nice horizontal scanline (walls and floors are special cases where the lines-of-constant-z are vertical and horizontal, respectively).

This diagonal (or free-direction) interpolation causes three major problems. First, if your diagonal lines don't abut properly you'll get dropouts and overwrites inside your polygon. This is solvable if you're careful.

Second, unless you obey a strict fill convention, you'll get the same kind of dropouts and overwrites between polygons. This is much harder to fix than the intra-polygon problems, but it's still solvable.

Finally, and this one is the kiss of death as far as I'm concerned, it is totally impossible to achieve subpixel accuracy with a lines-of-constant-z texture mapper. To achieve subpixel accuracy we must always sample the texture from the pixel centers of the destination but our arbitrary line-of-constant-z doesn't hit the pixel center in the destination. However, you can't step off the line-of-constant-z to the pixel center or you'll need to divide to take the nonconstant-z step into account. Damned if you do, damned if you don't.

If you don't care about subpixel accuracy (insert flame about jittering and sloppy textures here), the lines-of-constant-z technique might be for you. As an added bonus, you get depth cuing effects like fog almost for free—you already know the depth of the current line, and the depth, by definition, is going to stay constant. So you can compute your fog value at the start and use it along the entire line instead of at every pixel.

The next two techniques, subdividing affine and quadratic, are based on a more straightforward approximation of the perspective curve. Both try to fit easy-to-interpolate curves to the more complex perspective curve. Subdividing affine does a piecewise linear approximation, fitting a number of line segments to the curve, and quadratic fits a quadratic curve to the perspective curve.

We're going to use a subdividing affine curve for our approximation, so before going into detail on it I'll go over the quadratic technique.

Quadratics

Most people remember quadratic equations as parabolas from algebra. A quadratic in x is:

$$f(x) = ax^2 + bx + c \qquad (1)$$

This equation will graph a parabola or a line on the x and f(x) axes. The coefficients a, b, and c determine the

If you don't care about subpixel accuracy, the lines-of-constant-z technique might be for you.

shape and position of the parabola on the graph, and we use these three "degrees of freedom" to attempt to fit a parabola to the perspective curve. We use the normal perspective mapper to interpolate down the edges of the polygon so they'll be precise, and use the quadratic to interpolate across the scanline (where we're spending our time in `DrawScanLine`). At each pixel on the scanline, we want to be able to feed in our screen position, x, and the quadratic equation should produce our texture coordinate as its value (f(x) in Equation 1). We need two quadratics—one to produce the u texture coordinate from x, and the other to produce the v coordinate from x.

Because we have three degrees of freedom, we can choose to match exactly any three characteristics of the perspective curve, and approximate the other characteristics. For example, we might choose to exactly interpolate the two endpoints, and spend our last degree of freedom on exactly matching the first derivative, or slope, of the perspective curve as it enters or leaves one of the endpoints—we can't match both derivatives because that would cost us two degrees of freedom, one more than we have left if we're going to hit the endpoints exactly. We could even interpolate one endpoint and exactly match the first and second derivative exactly at that point. Or we might spend all three degrees of freedom interpolating three points on the curve exactly, like the two endpoints and the middle point. We'll do the derivation for the latter approximation to show how it's done.

To figure out the quadratic curve that interpolates the two endpoints and the midpoint and produces the u texture coordinate given x, we start by writing down the equations we know. We assume x goes from 0 to 1 for simplicity. We need to solve for u at x = 0, 0.5, and 1 using the perspective divide to give us three known values, u_0, u_1, and u_2, respectively, so we can solve for a, b, and c, the three unknowns. We plug these values into Equation 1 to give us three equations in three unknowns:

$$f(0) = u_0 = a0^2 + b0 + c = c$$

$$f\left(\frac{1}{2}\right) = u_1 = a\left(\frac{1}{2}\right)^2 + b\frac{1}{2} + c = \frac{a}{4} + \frac{b}{2} + c$$

and:

$$f(1) = u_2 = a1^2 + b1 + c = a + b + c$$

We then solve the simultaneous equations for a, b, and c in terms of u_0, u_1, and u_2, and after a bit of algebra we get:

$$a = 2u_0 - 4u_1 + 2u_2$$

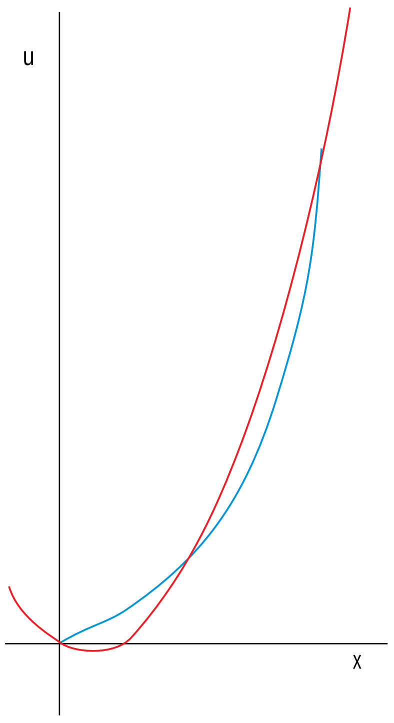
$$b = -3u_0 + 4u_1 - u_2$$

and:

$$c = u_0$$

Now, if we wanted to, we could use these coefficients and solve the quadratic directly at each point on the scanline (we'd actually do the math using x = 0, at x = width/2, and x = width so we wouldn't have to scale our x values between 0 and 1), but that means we'd be doing a bunch of multiplies per pixel—probably better than the divide we're currently doing, but not great. However, we can use forward differences to solve the quadratic using only addition and the solution for the previous pixel. Forward differences are covered in any good graphics or math textbook, but, simply stated, you take f(x+1)-f(x) to calculate the function's step based on its previous value. In the case of our qua-

Figure 2. The Quadratic Curve



Listing 2. The Subdividing Affine `DrawScanLine` (Continued on p. 24)

```
void DrawScanLine_suba( dib_info const &Dest,
    gradients_fx_fl_a const &Gradients,
    edge_fx_fl_a *pLeft, edge_fx_fl_a *pRight,
    dib_info const &Texture )
{
    int XStart = pLeft->X;
    int Width = pRight->X - XStart;

    char unsigned *pDestBits = Dest.pBits;
    char unsigned * const pTextureBits = Texture.pBits;
    pDestBits += pLeft->Y * Dest.DeltaScan + XStart;
    long TextureDeltaScan = Texture.DeltaScan;

    int const AffineLength = 8;

    float OneOverZLeft = pLeft->OneOverZ;
    float UOverZLeft = pLeft->UOverZ;
    float VOverZLeft = pLeft->VOverZ;

    float dOneOverZdXAff = Gradients.dOneOverZdX * AffineLength;
    float dUOverZdXAff = Gradients.dUOverZdX * AffineLength;
    float dVOverZdXAff = Gradients.dVOverZdX * AffineLength;

    float OneOverZRight = OneOverZLeft + dOneOverZdXAff;
    float UOverZRight = UOverZLeft + dUOverZdXAff;
    float VOverZRight = VOverZLeft + dVOverZdXAff;

    float ZLeft = 1/OneOverZLeft;
    float ULeft = ZLeft * UOverZLeft;
    float VLeft = ZLeft * VOverZLeft;

    float ZRight, URight, VRight;
    fixed16_16 U, V, DeltaU, DeltaV;

    if(Width > 0) {
        int Subdivisions = Width / AffineLength;
        int WidthModLength = Width % AffineLength;

        if(!WidthModLength) {
            Subdivisions--;
            WidthModLength = AffineLength;
        }

        while(Subdivisions-- > 0) {
            ZRight = 1/OneOverZRight;
            URight = ZRight * UOverZRight;
            VRight = ZRight * VOverZRight;

            U = FloatToFixed16_16(ULeft) + Gradients.dUdXModifier;
            V = FloatToFixed16_16(VLeft) + Gradients.dVdXModifier;
            DeltaU =
                FloatToFixed16_16(URight - ULeft)/AffineLength;
            DeltaV =
                FloatToFixed16_16(VRight - VLeft)/AffineLength;

            for(int Counter = 0;Counter < AffineLength;Counter++){
                int UInt = U>>16;
                int VInt = V>>16;

                *(pDestBits++) = *(pTextureBits + UInt +
                        (VInt * TextureDeltaScan));

                U += DeltaU;
                V += DeltaV;
            }

            ZLeft = ZRight;
            ULeft = URight;
            VLeft = VRight;
```

Listing 2. The Subdividing Affine `DrawScanLine` (Continued from p. 22)

```
            OneOverZRight += dOneOverZdXAff;
            UOverZRight += dUOverZdXAff;
            VOverZRight += dVOverZdXAff;
        }

        if(WidthModLength) {
            ZRight = 1/(pRight->OneOverZ - Gradients.dOneOverZdX);
            URight = ZRight *
                (pRight->UOverZ - Gradients.dUOverZdX);
            VRight = ZRight *
                (pRight->VOverZ - Gradients.dVOverZdX);

            U = FloatToFixed16_16(ULeft) + Gradients.dUdXModifier;
            V = FloatToFixed16_16(VLeft) + Gradients.dVdXModifier;

            if(--WidthModLength) {
                // guard against div-by-0 for 1 pixel lines
                DeltaU =
                    FloatToFixed16_16(URight - Uleft)
                    / WidthModLength;
                DeltaV =
                    FloatToFixed16_16(VRight - Vleft)
                    / WidthModLength;
            }

            for(int Counter = 0;
                Counter <= WidthModLength;Counter++) {
                int UInt = U>>16;
                int VInt = V>>16;

                *(pDestBits++) = *(pTextureBits + UInt +
                        (VInt * TextureDeltaScan));

                U += DeltaU;
                V += DeltaV;
            }
        }
    }
}
```

dratic, we need to do "second forward differences," where we calculate the forward difference of the function step. In other words, we calculate the forward difference of the forward difference.

How does it look? Well, in Figure 2, the blue curve is again the perspective curve, and the red curve is the quadratic interpolating the start, middle, and end. This is a pretty bad case for the quadratic because the perspective warp is quite high. On less distorted views the single quadratic would match up better. You can also subdivide into multiple quadratics to better match the curve if you want to spend the extra setup time. However, I chose this view because it illustrates a very significant side effect of the quadratic approximation—undershoot. If you look very closely, you'll see the red line actually dips below u = 0, which means we'd read off our texture map and possibly crash. You can figure out when this will happen and prevent it by subdividing, but that means even more setup in addition to setting up the quadratic equation for both u and v for each scanline.

Overall, the quadratic approximation is very elegant conceptually, but the problems of under- and overshoot and the setup overhead of calculating the coefficients seem to make it not worth the trouble. You can also use higher order curves, like cubics, and the math we've looked at extends easily. Perhaps we'll return to quadratics in a later column and see what we can do with them, but for now, we'll move on to subdividing affine.

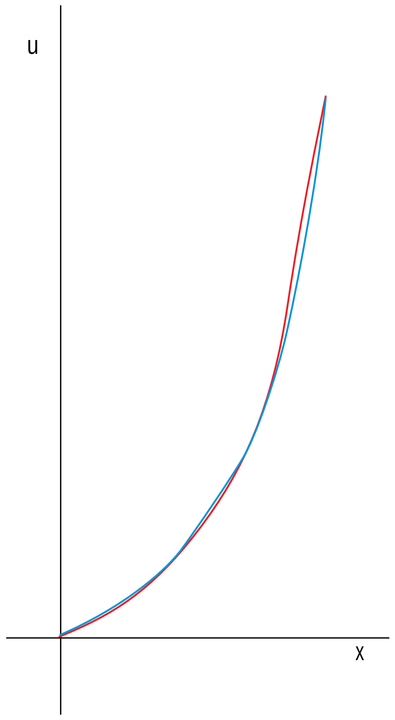
It's Affine Day

It happens that the method we've chosen is also the simplest. You've probably already guessed exactly how subdividing affine texture mappers work from what I've been describing.

Basically, you solve the real perspective equation at a bunch of points along the scanline, and do a linear interpolation between those correct points (affine and linear are virtually interchangeable in this context). Linear interpolations are what we've been doing all along with 1/z, u/z, and v/z, so I won't go into detail here. Linear interpolations are very fast, and the setup isn't too bad for each affine span. The only trick is to determine how often to subdivide; the more you subdivide, the closer your piecewise linear curve will match the real curve—but the more overhead you'll have from divides and per-span setup.

One simple way to subdivide is to just break up the scanline into equisized spans. You can also adaptively subdivide based on the amount of perspective warp on each span. This issue's texture mapper will always subdivide to eight pixel spans, but we'll look into adaptive subdivision next time. Figure 3 shows a subdividing affine approximation to our

Figure 3. The Subdivided Affine Curve



favorite perspective curve, subdividing every eight pixels in x.

There are a few nice things about subdividing affine texture mappers. First, you can tune the performance and quality by setting the subdivision level. This lets you adjust your performance on demand (which you might need to do at runtime, depending on scene complexity).

Second, you can pretty easily fit all your interpolants in registers for the affine spans (a subject I'll cover in more depth in my next column).

Finally, unlike quadratic approximation, you'll never under- or overshoot with subdividing affine because like the real perspective curve, the affine spans are monotonically increasing or decreasing with the curve. In other words, depending on your subdivision granularity and the perspective warp, you'll sometimes draw the wrong pixels, but you'll never fetch outside the texture map or even outside the extents of the original correct span in texture space.

Sample Code

Listing 2 shows the `DrawScanLine` function modified to do subdividing affine texture mapping. We'll max this out next time, but even unoptimized it outperforms the divide-per-pixel routine by two to four times. We must treat the last span with care to ensure we interpolate the rightmost pixel correctly. You'll remember from previous articles that our right edge is actually the left edge of the next polygon over, so we need to subtract one pixel from the right edge to figure out the last pixel in *our* polygon and use it in the interpolation.

I've also finally written a texture mapping test bed so you can simply compile and run the listings. You can find it on ftp://ftp.mfi.com/gdmag. The test has all the texture mappers we've written so far, so you can see the jitter from the integer mapper, the mapping bug from the one we discussed in the last installment, and so on. It's a Windows program, but the code for the texture mappers is portable, and you'll be able to see exactly how they're called. The test bed is easy to modify—see the readme.txt file in the archive.

In parting, I want to mention a few tidbits. First, you might think you should do an approximation down the edges as well, which is certainly possible. However, your error can build up pretty quickly if you're not careful. Also, *Digital Image Warping* by George Wolberg (IEEE Computer Society, 1990) is a pretty good reference for this sort of thing (including forward differences). Finally, I'd like to thank Chris Green from Leaping Lizard Software for opening my eyes up to the fact that a, b, and c really are three totally arbitrary degrees of freedom. ■

By the time you read this, Chris Hecker will have quit working for the man and will be out on his own, finally paying for his own beverages. You can recommend your favorite drink at checker@bix.com.