Attention:

Chris Hecker
definition six, incorporated
checker@d6.com
http://www.d6.com/users/checker

PS. The email address at the end of the article is incorrect. Please use checker@d6.com for any correspondence.

Perspective Texture Mapping, Part III: Endpoints and Mapping

Chris Hecker

If you think we've covered everything on perspective texture mapping, you're wrong. In Part III of this ongoing series, we get a close look at the math involved in endpoints and mapping.

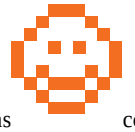
By the time you read this article, the Electronic Entertainment Expo (E3) will be long over, but in the time warp of magazine article submission deadlines it was just last weekend in Los Angeles. E3 is the game industry's attempt to break from the huge toaster, car stereo, and microwave oven event that is the Consumer Electronics Show. Whether this breakaway was successful remains to be seen, but one thing is certain: the new generation of video game consoles garnered a lot of attention and floorspace. Atari, Sega, 3DO, and Sony battled for developers' attention, each hoping to wow people with its machine's high-end features and get the really cool games developed for its platform—the Jaguar, the Saturn, the Multiplayer, and the PlayStation, respectively.

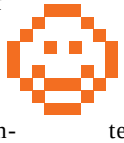
The reason I bring this to your attention is the one feature advertised above all others for each machine is—you guessed it—texture mapping. Each company claims its system has the most realistic texture mapping, or the fastest texture mapping, or the least expensive texture mapping.

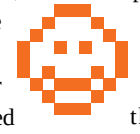
I'll mention one important caveat before I lay into this generation of hardware with technical criticism. It's completely unclear what relation, if any, exists between texture mapping quality and overall game quality (and certainly sales). Super Mario Bros., for example, has absolutely no texture mapping, but it sure is a great game, both from a playability and profitability standpoint.

Keeping that in mind, the texture mapping on these machines sure does suck.

How do they screw up texture mapping? Let me count the ways. First and most noticeable is that all the texture mapping hardware in this generation is affine. Affine texture mapping, as we discussed in "Perspective Texture Mapping Part I: Foundations" (Under the Hood, April/May 1995), assumes the equation to map screen coordinates to texture coordinates is linear. This results in really nasty texture warping when the linear equation and the true equation start to differ by a substantial amount. The ironic part about affine texture mapping is these two equations differ most when the textures are very close to your viewpoint, which makes the problem easy to spot.







You can clearly see this for yourself in almost every game produced for these machines. Check out the floors in some of the fighting games or the walls in walkthrough or driving games. Get real close and prepare for a stomach-churning texture dance.

Second, and particularly germane to today's discussion, some of these machines only support integer-texture coordinates, that is, the vertices of the polygons can only correspond to integer coordinates in the source bitmap. This wouldn't seem so bad until you realize one of the ways to combat the affine problems I've mentioned is to subdivide your polygons until the linear equation is a closer fit (we'll cover this technique in the near future). The subdivision points are not likely to fall on integer texture coordinates, so this hardware

forces you to snap to the nearest integer, resulting in jitter that's plainly visible in the games.

Finally, a few of the machines only support integer screen-space polygon vertices. In other words, if your polygon comes out of your three-dimensional transform pipeline with noninteger endpoints (as it's very likely to do) you've got to snap the vertex to an integer pixel location, which causes even more jitter. Conveniently for the purposes of this article, this is the exact jitter problem we introduced into our own texture mapper when we converted from floating-point to integer rasterization ("Perspective Texture Mapping, Part II: Rasterization," Behind the Screen, June/July 1995). Of course, we haven't spent millions of dollars on developing a piece of hardware and marketing it, so we can fix our jitter problem pretty easily.

Jitter Bug

We don't have the space to do a total review of the work we covered in my first two columns on texture mapping. However, the five-second summary is as follows. In the first column, we derived the perspective texture mapping equations, including the equations for perspective projection and those for stepping the texture coordinates across the destination polygon (these step values are called the gradients). We also looked at how to correctly sample with subpixel accuracy. This last topic caused us to investigate how to get rid of the cost of this subpixel accuracy while retaining its advantages, and in the second column we showed how to do this using a digital differential analyzer (DDA). We converted our original floating-point rasterizer to an integer DDA to realize the savings, but we uncovered a nasty jitter as our polygon moved and animated.

This jitter was introduced because our triangle gradients are calculated from the endpoints of the triangle, and those endpoints, when restricted to be integers, change by a relatively large amount from frame to frame. (The mathematically inclined among you will notice that the gradients are calculated from two times the signed area of the triangle [which is also the cross product]. When the endpoints are truncated to integers this area changes, altering our gradients and causing the jitter.)

What we need is better precision on the endpoints, but we want to keep the advantages of using a DDA rasterizer. Enter fractional endpoints.

Figure 1. Equations 1 Through 3

$$x_{\text{int}} = \left\lceil \frac{x_1 - x_0}{y_1 - y_0}\left(y - y_0\right) + x_0 \right\rceil \qquad (1)$$

$$\left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{a-1}{b} \right\rfloor + 1 = \left\lfloor \frac{a-1}{b} + 1 \right\rfloor = \left\lfloor \frac{a-1+b}{b} \right\rfloor \qquad (2)$$

$$\frac{a}{b} = \left\lfloor \frac{a}{b} \right\rfloor + \frac{a \bmod b}{b} \qquad (3)$$

Fractional Endpoints

When I say fractional endpoints, the first thing that comes to mind is fixed-point math. While we are going to be using fixed-point numbers to represent our vertices and to give us the extra precision we need to avoid the integer jitter, you'll see we're not going to be rasterizing the edges using the familiar fixed-point increments. As usual, to pack all the information we need into this article, I'm not going to be able to describe the basics of fixed-point math. For a description of fixed-point math that's easy to understand, I suggest reading Michael Abrash's *Zen of Graphics Programming* (Coriolis Group, 1994) or the *PC Game Programmer's Encyclopedia*, which is a neat freeware programming book available via ftp on x2ftp.oulu.fi.

We'll use 28.4 fixed point for our endpoints. I'm going to use the integer-dot-fraction notation for fixed-point numbers, so 28.4 means we have 28 bits of (usually signed) integer and four bits of fractional precision.

We'll use this format for two reasons. First, four fractional bits is enough to eliminate the jitter. Second, I happen to know that the Windows NT polygon rasterizer can be set up to do correct top-left 28.4 rasterization, and it always helps to have a proven version against which to test (although I won't show it here, we can write a program that rasterizes a polygon with our code, then rasterizes the polygon with Windows NT's rasterizer, so we can check for differences to test our rasterizer). Once you see how the math works you'll be able to derive a rasterizer for whatever fixed-point format you like best.

As I hinted before, instead of using a fixed-point or floating-point incremental step to move from one scanline to the next, as our first rasterizer did, this rasterizer will use an error-term DDA (much like the Bresenham line-drawing algorithm, covered in most graphics books). However, unlike most DDA rasterizers you've probably seen, our DDA parameters will be initialized with fixed-point numbers instead of integers.

We'll start by defining exactly what we mean by fractional endpoints. From here on out, x and y are real numbers, not integers, and their values are m/F and n/F, respectively. The numbers m and n are integers, and F is the scaling

Figure 2. Equation 4

$$x_{\text{int}} = \left\lfloor \frac{F\mathrm{D}my - \mathrm{D}mn_0 + \mathrm{D}nm_0 - 1 + F\mathrm{D}n}{F\mathrm{D}n} \right\rfloor \qquad (4)$$

factor for whatever fixed-point format you're using. For 28.4 fixed-point, F = 16, for 16.16, F would be 65,536, and so on. To convert from the fixed-point values to real numbers we divide by the scaling factor. You commonly see the opposite of this when you multiply a floating point number by the scaling factor to get its fixedpoint value. Here are some useful equations:

$$\Delta x = x_1 - x_0 = \frac{\Delta m}{F} = \frac{m_1 - m_0}{F}$$

$$\Delta y = y_1 - y_0 = \frac{\Delta n}{F} = \frac{n_1 - n_0}{F}$$

We'll be reusing some of the formulas we derived in the first two columns. Refer to Figure 1 for Equations 1 through 3. The variables a and b are integers in these equations—remember that the mathematically defined `mod` operator (used in Equation 3) probably behaves slightly differently than the modulus operator in your chosen programming language. See the `FloorDivMod` function in Listing 1 for the correct implementation and "Perspective Texture Mapping, Part II: Rasterization" for an in-depth discussion. Equation 1 shows the real formula for a left edge under our fill convention (right edges are the same equation minus one). Let's rewrite Equation 1 to use fixed point:

$$x_{\text{int}} = \left\lceil \frac{\frac{\Delta m}{F}}{\frac{\Delta n}{F}} y - \frac{n_0}{F} \right\rceil + \frac{m_0}{F}$$

If we do some basic algebra and use the ceiling-to-floor conversion in Equation 2 (you can move integers into and out of a ceiling or floor) on this we get the equation pictured in Figure 2.

Next we'll introduce the symbol R (Why R? I don't know, mostly because I'm running out of letters in the alphabet!), set it equal to the numerator so things look pretty, and finally use Equation 3 (the relationship between a rational number and its floor and `mod`) on the R/FΔn term inside the floor to give us our initial condition:

Listing 1. Changes for Fractional Endpoints (Continued on p. 21)

```
typedef long fixed28_4;
inline fixed28_4 FloatToFixed28_4( float Value ) {
    return Value * 16;
}
inline float Fixed28_4ToFloat( fixed28_4 Value ) {
    return Value / 16.0;
}
inline fixed28_4 Fixed28_4Mul( fixed28_4 A, fixed28_4 B ) {
                        // could make this asm to prevent overflow
    return (A * B) / 16;    // 28.4 * 28.4 = 24.8 / 16 = 28.4
}
inline long Ceil28_4( fixed28_4 Value ) {
    long ReturnValue;
    long Numerator = Value - 1 + 16;
    if(Numerator >= 0) {
        ReturnValue = Numerator/16;
    } else {
        // deal with negative numerators correctly
        ReturnValue = -((-Numerator)/16);
        ReturnValue -= ((-Numerator) % 16) ? 1 : 0;
    }
    return ReturnValue;
}
struct POINT3D {
    fixed28_4 X, Y;
    float Z;
    float U, V;
};
inline void FloorDivMod( long Numerator, long Denominator, long &Floor,
              long &Mod )
{
    assert(Denominator > 0);                    // we assume it's positive
    if(Numerator >= 0) {
        // positive case, C is okay
        Floor = Numerator / Denominator;
        Mod = Numerator % Denominator;
    } else {
        // Numerator is negative, do the right thing
        Floor = -((-Numerator) / Denominator);
        Mod = (-Numerator) % Denominator;
        if(Mod) {
            // there is a remainder
            Floor--; Mod = Denominator - Mod;
        }
    }
}
gradients::gradients( POINT3D const *pVertices )
{
    int Counter;
    fixed28_4 X1Y0 = Fixed28_4Mul(pVertices[1].X - pVertices[2].X,
                                  pVertices[0].Y - pVertices[2].Y);
    fixed28_4 X0Y1 = Fixed28_4Mul(pVertices[0].X - pVertices[2].X,
                                  pVertices[1].Y - pVertices[2].Y);
    float OneOverdX = 1.0 / Fixed28_4ToFloat(X1Y0 - X0Y1);
    float OneOverdY = -OneOverdX;
    for(Counter = 0;Counter < 3;Counter++)
    {
        float const OneOverZ = 1/pVertices[Counter].Z;
        aOneOverZ[Counter] = OneOverZ;
        aUOverZ[Counter] = pVertices[Counter].U * OneOverZ;
        aVOverZ[Counter] = pVertices[Counter].V * OneOverZ;
    }
    dOneOverZdX = OneOverdX * (((aOneOverZ[1] - aOneOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].Y - pVertices[2].Y)) -
            ((aOneOverZ[0] - aOneOverZ[2]) *
            Fixed28_4ToFloat(pVertices[1].Y - pVertices[2].Y)));
    dOneOverZdY = OneOverdY * (((aOneOverZ[1] - aOneOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].X - pVertices[2].X)) -
            ((aOneOverZ[0] - aOneOverZ[2]) *
```

$$R = F\Delta m y - \Delta m n_0 + \Delta n m_0 - 1 + F\Delta n$$

$$x_{int} = \left\lfloor \frac{R}{F\Delta n} \right\rfloor + \frac{R \bmod F\Delta n}{F\Delta n} = \left\lfloor \frac{R}{F\Delta n} \right\rfloor + \left\lfloor \frac{R \bmod F\Delta n}{F\Delta n} \right\rfloor \quad (5)$$

Notice that we moved the floored R/FΔn term outside the main floor; we can do this because a floored term is an integer by the definition of the floor function, and you can always move an integer into and out of a floor.

Equation 5 is our initial DDA condition. If we plug in integer y values and do the math correctly, the floored R/FΔn term will be our initial integer x starting location, and the numerator of the **mod** term will be our initial DDA error term. Before we plug it into this equation, our y should be prestepped to the first scanline according to our fill convention, which defines the starting integer y as the ceiling of the fractional y. Alternatively, if you're two-dimensionally clipping the polygon at rasterization time, you'd make y be the first scanline you want to draw after clipping.

To calculate our DDA step variables for x to step to x', we plug y = y + 1 into Equation 4 and see that our equation changes by FΔm/FΔn. We use Equation 3 to convert this ratio into an integer and a fractional part:

$$x'_{int} = x_{int} + \left\lfloor \frac{F\Delta m}{F\Delta n} \right\rfloor + \left\lfloor ErrorTerm + \frac{F\Delta m \bmod F\Delta n}{F\Delta n} \right\rfloor \quad (6)$$

Equations 5 and 6 will step us on the integer raster grid, but we will step according to the fractional edge, so we'll get the extra precision. As I've mentioned before, it's important to notice that the **mod** terms are always positive, so when our error term rolls over our DDA will always step by 1 (in contrast with some other DDAs you've probably seen where you step by 1 for right-going edges and by -1 for left-going edges). This slightly odd behavior drops out of the math when you do the flooring divide and **mod** correctly, as we discussed previously.

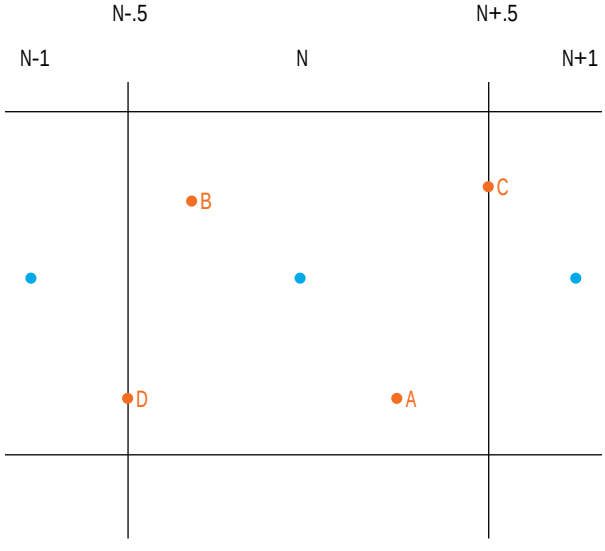
We're still doing the same DDA step as in my last column (the step code is identical), but the various DDA values are determined by the real fractional endpoints, not by the truncated integers. More importantly, the gradients are calculated with the fractional endpoints, which avoids the jitter problems that brought up this fractional mess in the first place.

The results are surprising. Visually, you can't tell the difference between our original floating-point rasterizer and the new fractional endpoint rasterizer—they both are completely solid and jitter-free—

Listing 1. Fractional Endpoints (Continued from p. 20)

```
            Fixed28_4ToFloat(pVertices[1].X - pVertices[2].X)));
    dUOverZdX = OneOverdX * (((aUOverZ[1] - aUOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].Y - pVertices[2].Y)) -
            ((aUOverZ[0] - aUOverZ[2]) *
            Fixed28_4ToFloat(pVertices[1].Y - pVertices[2].Y)));
    dUOverZdY = OneOverdY * (((aUOverZ[1] - aUOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].X - pVertices[2].X)) -
            ((aUOverZ[0] - aUOverZ[2]) *
            Fixed28_4ToFloat(pVertices[1].X - pVertices[2].X)));
    dVOverZdX = OneOverdX * (((aVOverZ[1] - aVOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].Y - pVertices[2].Y)) -
            ((aVOverZ[0] - aVOverZ[2]) *
            Fixed28_4ToFloat(pVertices[1].Y - pVertices[2].Y)));
    dVOverZdY = OneOverdY * (((aVOverZ[1] - aVOverZ[2]) *
            Fixed28_4ToFloat(pVertices[0].X - pVertices[2].X)) -
            ((aVOverZ[0] - aVOverZ[2]) *
            Fixed28_4ToFloat(pVertices[1].X - pVertices[2].X)));
}
edge::edge( gradients const &Gradients, POINT3D const *pVertices,
            int Top, int Bottom )
{
    Y = Ceil28_4(pVertices[Top].Y);
    int YEnd = Ceil28_4(pVertices[Bottom].Y);
    Height = YEnd - Y;
    if(Height)
    {
        long dN = pVertices[Bottom].Y - pVertices[Top].Y;
        long dM = pVertices[Bottom].X - pVertices[Top].X;
        long InitialNumerator = dM*16*Y - dM*pVertices[Top].Y +
                    dN*pVertices[Top].X - 1 + dN*16;
        FloorDivMod(InitialNumerator,dN*16,X,ErrorTerm);
        FloorDivMod(dM*16,dN*16,XStep,Numerator);
        Denominator = dN*16;
        float YPrestep = Fixed28_4ToFloat(Y*16 - pVertices[Top].Y);
        float XPrestep = Fixed28_4ToFloat(X*16 - pVertices[Top].X);
        OneOverZ = Gradients.aOneOverZ[Top]
                        + YPrestep * Gradients.dOneOverZdY
                        + XPrestep * Gradients.dOneOverZdX;
        OneOverZStep = XStep * Gradients.dOneOverZdX
                        + Gradients.dOneOverZdY;
        OneOverZStepExtra = Gradients.dOneOverZdX;
        UOverZ = Gradients.aUOverZ[Top]
                        + YPrestep * Gradients.dUOverZdY
                        + XPrestep * Gradients.dUOverZdX;
        UOverZStep = XStep * Gradients.dUOverZdX
                        + Gradients.dUOverZdY;
        UOverZStepExtra = Gradients.dUOverZdX;
        VOverZ = Gradients.aVOverZ[Top]
                        + YPrestep * Gradients.dVOverZdY
                        + XPrestep * Gradients.dVOverZdX;
        VOverZStep = XStep * Gradients.dVOverZdX
                        + Gradients.dVOverZdY;
        VOverZStepExtra = Gradients.dVOverZdX;
    }
}
```

Figure 3. A Pixel



and we get all the benefits of doing error-term integer DDAs.

Listing 1 shows the changes to our texture mapper to use fractional endpoints. One thing to watch for is overflow in these equations, particularly in the numerator of Equation 4. If your polygons get really big, and your scaling factor is large, you can overflow beyond 32 bits. Most architectures make it possible to keep a 64-bit numerator around for the divide, so you can usually handle this if the need arises.

Off The Map

Let me just come out and say it: there's a bug in the code from my first column on perspective texture mapping. No, it's not a bug in any of the rasterization math or implementation we've been poring over for the last two issues, and it's subtle enough that you'd have to know what you were looking for and look pretty hard to find it. In fact, Michael Abrash and I were talking about a related issue when we realized there actually was a bug in the code and math. We even tossed around the idea of having a contest to spot the bug, but I decided against it because I assumed it was so subtle nobody would figure it out. Of course, within the next day or so Walt Donovan (walt@rendition.com) from Rendition Inc. sent me e-mail describing the very problem!

The bug is in the only part of the code where I didn't rigidly define the math before I started out: the real-to-integer source-texture coordinate mapping.

As we've seen, we have rock-solid mathematical descriptions of the rasterization, the subpixel stepping, the perspective projection, and the gradient calculations. But when it comes time in the code to take our real texture coordinates for the current pixel center and map them into integer-source-texture coordinates, we simply truncate with no explanation of whether this is the correct thing to do or not. It's not, and we're going to figure out why. Here's the suspect code from our original `DrawScanline` function (the variables on the right are floating point numbers):

```
int U = UOverZ * Z;
int V = VOverZ * Z;
```

To understand why this code is wrong, we need to understand how the mapping from the source to the destination (or vice versa) works, and to understand this, we need to understand the lowliest element in the graphics pipeline, the pixel. As we hinted in previous issues, a pixel isn't a single point as we're used to thinking, it's really a box; a small box, but a box nonetheless. Like every other box (with sides of nonzero length), this one covers an area, and we need to take that area into account when we texture map our polygon.

Figure 3 shows one complete pixel and portions of a couple of its neighbors. We'll call N our integer pixel coordinate, and you can see the edges, or walls, of the middle pixel are each a half-pixel away from the center. This geometry gives our pixel a total area of one, as you'd expect. The other pixel centers are exactly one unit away on either side.

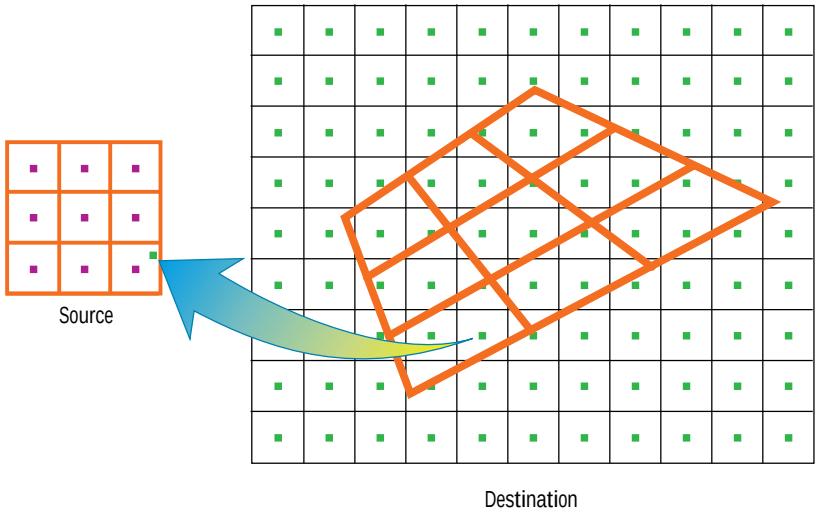
As we rasterize our polygon on the destination grid, we're very careful to only light destination pixels when they're "in," according to our precisely defined fill convention, and we're also very careful to only generate source texture coordinates (using the truncation code for the time being) when we're exactly on a destination pixel center. We're basically projecting the destination pixel center back into the source to figure out the source pixel color with which to light the destination pixel. The `UOverZ * Z` expression generates this real source texture coordinate, and our "mapping rule," such as it is, converts this real number into an integer source coordinate we can use to look up the texture pixel (sometimes called a texel) value.

Figure 4, which shows the source texture and its position on the destination, gives us a way of visualizing the problem. You can also see each of the source texel boundries drawn in the destination for the purposes of this illustration. If we were to rasterize this destination polygon to texture map our source, we'd generate source coordinates for each of the destination pixel centers. As you can see (with the help of the arrow showing one of the destination pixel centers mapping back into the source) those pixel centers rarely, if ever, map to the source pixel centers.

Let's look at how our current truncation mapping rule affects various source coordinates by viewing the pixels in Figure 3 as the source texels. If our perspective projection for a given mapping takes a destination pixel center and maps it to the real point denoted by the A in Figure 3, our truncation rule maps this to pixel N (A's value is greater than N, but less than N + 1, so truncation maps it to N), which looks about right. In general, we'd like our mapping rule to take the real source coordinates and map them to the nearest integer pixel center, which means any points that fall within the box for a given source pixel get mapped to that pixel's integer coordinate.

Next let's say our projection takes our destination pixel center to B in Figure 3.

Figure 4. Mapping



Our truncation generates N - 1 for our source coordinate, when N is clearly the right answer! Oops.

Now that you see the bug, let me tell you how we figured out it was there in the first place. We were talking about various texture mapping one day when Michael asked how I would implement a `blt` (a block transfer, or pixel copy) with my texture mapper. In other words, how would I allocate the source-texture coordinates and the destination-screen coordinates so that the source-to-destination mapping was one to one? It seems logical to be able to do this, not so much because you'll use the texture code when you simply want to `blt`, but because if the math is completely right you should be able to get an exact 1:1 mapping just like all the other arbitrary mappings you can get with perspective projections.

I thought about this problem for a second, and answered that I'd allocate the corner texture coordinates and the destination coordinates at the exact same coordinates on the screen, (-0.5,-0.5) and (TextureWidth-0.5,TextureHeight-0.5). (Our code only handles triangles so I'd obviously need to call it twice to `blt` the whole source, but the top-left and bottom-right corners are all I needed to describe the destination rectangle.) As soon as I said this I realized I hadn't bothered to define the mapping rule from real source coordinates to integer source coordinates.

Figure 5 helps illustrate why I chose the coordinates I did. I can't stress enough in this discussion that to get the correct mappings we need to view pixels as areas, not just as points. With this in mind, the coordinates I gave Michael are the coordinates of the inifinitely thin edge that completely surrounds our source texture bitmap, as you can see in Figure 5. The point (-0.5,-0.5) is the upper left-hand corner of the texture (the upper left-hand pixel center is at [0,0]), and (9.5,9.5) is the lower right-hand corner—the edges totally enclose the texture pixels. If I had chosen integer source coordinates for the corners we'd be cutting the edge pixel areas in half, which you can see if you take Figure 5 and draw an imaginary edge through the pixel center.

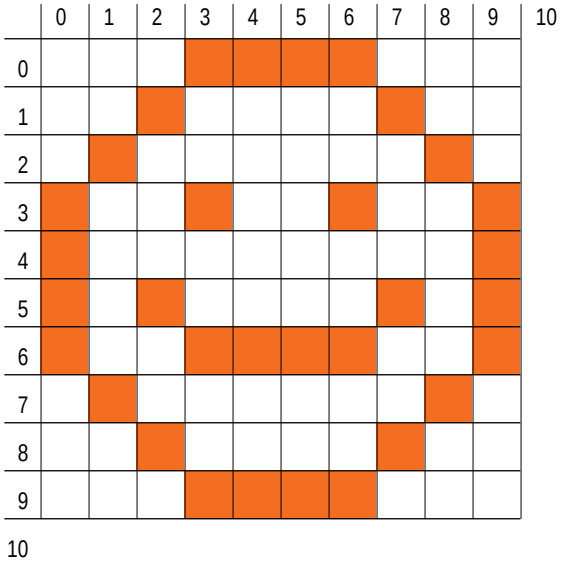
Similarly, I chose the corresponding screen coordinates for the destination. I wanted the destination pixel centers to map exactly to the source pixel centers, so it was necessary to completely enclose the destination pixels in the same manner as the source pixels.

I realized I needed to define a mapping that took a real texture coordinate and mapped it to the closest integer pixel center. This is basically a rounding operation, and the function for rounding is:

$$C_{\text{int}} = \left\lfloor C + \frac{1}{2} \right\rfloor \qquad (7)$$

Equation 7 is the familiar rounding rule where you add a half and floor the result. I looked at what effect this rule would have on the texture mapper and it fixes the problem with B in Figure 3, that's for sure. However, implicit in any rounding rule is a tie-breaking rule that kicks in when the value to be rounded is exactly halfway between two integers, like C and D in Figure 3. Equation 7 will map C to N + 1 and D to N, so it's pretty clear that this is a

Figure 5. Source Texture



top-left rounding rule, meaning if the pixel center falls on the top or left edge of the pixel it is considered "in," and if it falls on the bottom or right edge it is considered part of the neighboring pixel. This is obviously very similar to our fill convention.

At first glance, a top-left rounding rule seems to work well with our top-left fill convention. It looks like the only way for our mapping rule to generate a pixel that's out of bounds is for the right or bottom edge of the texture to fall directly on a pixel center in the destination (think about shifting our `blt` destination coordinates down and to the right by a half pixel so they're on integers in the destination), but if the right or bottom edge of the texture corresponds to a right or bottom edge of the destination polygon we don't draw those pixel centers anyway because of our fill convention. This beautiful harmony is broken when you visualize rotating the destination polygon by 180 degrees so that its edges still correspond with integer destination pixel centers, but the left edge of the polygon corresponds to the right edge of the texture. Now, if we apply our rounding mapping rule we'll start with the right edge of the texture (TextureWidth - 0.5, or 9.5 in Figure 3), add a half, and floor, resulting in a texture coordinate off the edge of our texture! Are we back where we started?

Same Time, Same Channel

The answer to that question is no, but to find out how we're going to solve the problem you'll have to tune in next time because I'm out of space. I will give you a hint, however. Equation 7 is just one rounding rule. Here's another:

$$C_{\text{int}} = \left\lfloor C - \frac{1}{2} \right\rfloor \qquad (8)$$

Equation 8 works out to be a bottom-right rounding rule, which would have avoided the problem mentioned at the end of the last paragraph, but would result in a similar problem with our original orientation! Think about what criteria we have for choosing between the two rounding rules and join me next issue.

By the way, our texture mapper is doing a form of resampling called *point sampling*. We map the destination pixel centers into the source and just take whatever texel in which we land. There are other forms of resampling where you take the corners of the destination pixel and map them back into the source to form a quadrilateral, and then filter the resulting area into a single pixel color. *Digital Image Warping* (IEEE Computer Society, 1990) by George Wolberg covers a bunch of these resampling techniques.

Once again I'd like to thank Kirk Olynyk. He's the reason I know the Windows NT 28.4 rasterizer is correct—he did the original math. ■

Chris Hecker thinks that regardless of the outcome of the video game console wars, it's unlikely anyone will beat Sega's "sphincter" advertisement for pure comedy value. Discussion of various body parts and their relationship to video games is available at checker@bix.com.