Attention:

This material is copyright © 1995-1997 Chris Hecker. All rights reserved.

You have permission to read this article for your own education. You do not have permission to put it on your website (but you may link to my main page, below), or to put it in a book, or hand it out to your class or your company, etc. If you have any questions about using this article, send me email. If you got this article from a web page that was not mine, please let me know about it.

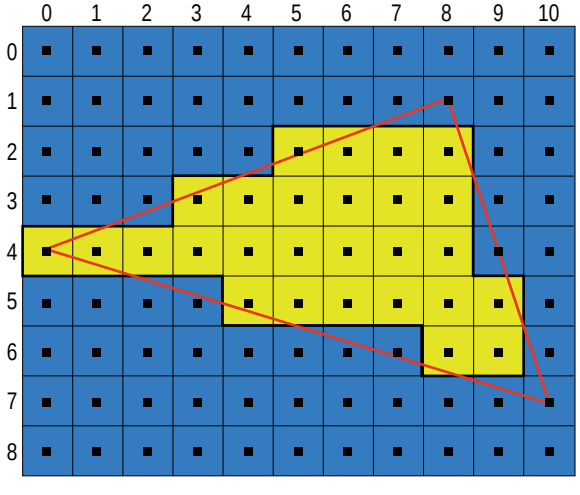
Thank you, and I hope you enjoy the article,

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 II: Rasterization

Figure 1. The Fill Convention



Did I say I'd be doing two columns? Silly me—I meant four or five columns. Our topic, perspective texture mapping, is so huge I don't know what I was thinking when I said we could cover it completely in two columns. Luckily, the topic has enough variety that it should keep everyone glued to these pages for the duration.

In Part 1, we covered most of the math behind the perspective projection and triangle gradients (those neat numbers that let us interpolate without recalculating at each scanline), and we quickly went over polygon fill conventions and stepping on pixel centers. That's a lot of information for a single article. In fact, there's so much material still to cover I'm not even going to summarize my last article beyond saying, "Read it." If you haven't read Part I, you'll still get a lot out of Part II, but you might have trouble seeing how this information fits in perspective (cough).

This time around we're going to focus on the triangle rasterization stage, and we'll expand on the math for the fill convention we derived last issue.

As I did last time, I encourage you to get out a piece of graph paper and join in the fun. Speaking for myself, I find it impossible to learn math without scribbling all over the place.

If you don't like math, well, computer graphics is math for the most part, so I'm not sure what to tell you. My goal is to describe the math in an accessible way, but I'm not going to hide the fact that math underlies everything about computer graphics, especially three dimensional computer graphics. If you like programming you will definitely like math...heck, math's even better than computer programming because there are no compiler or operating system bugs! (Of course, there's no compiler or operating system to tell you when you've done something wrong, either.)

Raster Blaster

When I say rasterization, I mean taking the continuous geometric triangle—defined by its vertices—and displaying it on the monitor's discrete display grid, or "raster." The rule we defined for doing this is called a top-left fill convention, where we light all pixels that are strictly inside the polygon boundaries and any pixels that are exactly on the polygon boundary if they're on the top or left edges (remember, pixels are boxes with a center, not just points). Figure 1 shows this fill convention in

Chris Hecker

Perspective texture mapping is a huge subject—much too big to cover in one or even two articles. In Part II of his series on the subject, Chris Hecker tackles rasterization, an essential concept.

action. Pixel (5,2) is lit, but pixel (9,4) is not, even though our polygon edge intersects both (within the limits of the magazine's printing accuracy, at least). This is because (5,2) is on a left edge and (9,4) is on a right edge. A fill convention lets abutting polygons share an edge without either polygon overwriting any pixels of its neighbor, or leaving any unlit holes—called dropouts—between the two.

A top-left fill convention for a left edge from x0,y0 to x1,y1 is defined mathematically by the ceiling function:

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

In my last column, I presented this equation without much explanation, so this time we'll go into it in more detail. First, we can derive the equation for the line in Figure 2 by setting the slope of the entire line equal to the slope of any line segment on that line (the segment from x,y to x_0,y_0 is on the line, so its slope is equal to the line's) and solving for x:

$$\frac{y_1 - y_0}{x_1 - x_2} = \frac{y - y_0}{x - x_0}$$

$$x = \left(\frac{x_1 - x_0}{y_1 - y_0}\right)(y - y_0) + x_0 \qquad (2)$$

We can use Equation 2 to give us the x value for any y value on the line. You can see that if $y = y_0$, then $x = x_0$ as you'd expect, and likewise for the other endpoint. This equation generates real (as opposed to integer) values for x, so we need to use our fill convention to tell us how the real x maps to an integer pixel. This is where the ceiling function comes in.

The ceiling function is defined as bumping a real value up to the next highest integer if the value has a fractional part, or leaving it alone if it is already an integer. For example:

$$\left\lceil \frac{4}{4} \right\rceil = 1$$

$$\left\lceil \frac{3}{4} \right\rceil = 1$$

$$\left\lceil \frac{5}{2} \right\rceil = 3$$

$$\left\lceil \frac{-4}{4} \right\rceil = -1$$

and:

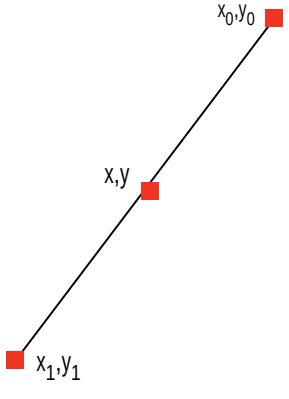
$$\left\lceil \frac{-4}{3} \right\rceil = -1$$

Notice how the ceiling behaves with negative numbers—it bumps the value to the next highest value, not to the next highest absolute value.

The ceiling is the perfect function to realize a top-left fill convention for left edges (and top edges if you solve for y instead of x in Equation 2). If we're exactly on an integer pixel center we will light the pixel, but if our x is at all greater than the integer—to the left of the pixel center—the ceiling will bump us up to the next pixel that's strictly inside the edge. It should be pretty obvious that the equations for right and bottom edges are the same as for top and left edges with the addition of a minus one outside the ceiling. That is, if the edge is on the integer pixel, the ceiling won't affect it, but the

minus one will knock us back one pixel following our fill convention. If the edge is greater than the pixel, the ceiling will bump it up one (to the first pixel outside the edge) and the minus one will bump it right back inside the polygon. Another way of looking at it is if we have two polygons with an abutting edge, the edge will be the left edge of one and the right edge of the other (or the top and the bottom), and they'll draw the same set of pixels,

Figure 2. A 2D Line



except offset by a single pixel for one polygon.

The code to implement this was pretty straightforward in our floating point rasterizer (shown in last month's listing):

```
int XStart = ceil(pLeft->X);
```

We call the ANSI standard math.h function, `ceil()`, and use the integer returned for our starting x coordinate. As we step from one scanline to the next our real x value steps by the inverse slope, as Equation 2 shows when you set y = y + 1.

While floating-point math certainly is convenient when you're trying to get code up and running, it's probably not the best choice for a production rasterizer. First, even though floating point coprocessors are commonplace on today's machines and are even faster than the integer processor for some operations, converting from floating point to integer is still slow. Because a rasterizer is where the real three-dimensional coordinates get mapped to the integer hardware bitmap, we end up converting a lot. Also, functions like `ceil()` are actual function calls in floating point, but fall out of the math almost for free with integer coordinates.

In addition, it's hard to get the math just right for floating point numbers; there's a whole field in mathematics dedicated to figuring out how floating point numbers accumulate error. Finally, we'll see there are some benefits to using integer digital differential analyzers (DDAs) when we discuss pixel centers.

Integers from Floor to Ceiling

Before we convert our rasterizer to use integers, let's learn a couple of neat tricks for manipulating the ceiling function and its companion, the floor. The floor of a value is—you guessed it—the next-lowest integer if the value has a fractional part, or the value if it's already an integer. You could also think of this as truncating the fractional part for positive values. Following are some floor examples:

$$\left\lfloor \frac{4}{4} \right\rfloor = 1$$

$$\left\lfloor \frac{3}{4} \right\rfloor = 0$$

$$\left\lfloor \frac{5}{2} \right\rfloor = 2$$

$$\left\lfloor \frac{-4}{4} \right\rfloor = -1$$

and

$$\left\lfloor \frac{-4}{3} \right\rfloor = -2$$

Again, notice the behavior when the value is negative.

We can convert from ceiling to floor easily if a and b are integers:

$$\left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a-1}{b} \right\rfloor + 1 = \left\lfloor \frac{a-1}{b} + 1 \right\rfloor = \left\lfloor \frac{a-1+b}{b} \right\rfloor \quad (3)$$

Equation 3 also shows that we can move integers in and out of the floor (or ceiling). We obviously can't move fractional values in and out, though, because they can affect the result. Run through a few examples on your own to see why Equation 3 works.

Now that we have a working knowledge of floors and ceilings, let's convert the rasterizer to use integer coordinates. Because we are defining x_0,y_0 and x_1,y_1 in Equation 1 to be integers, we can manipulate the equation to our advantage. We can bring x_0 outside the ceiling function, for starters. This means any x generated by our fill convention will be the integer x_0 plus the integer result of the ceiling function for a given y. Now, let's use Equation 3 to turn the ceiling function into a floor.

Let:

$$dx = x_1 - x_0$$

and:

$$dy = y_1 - y_0$$

so:

$$x_{\text{int}} = \left\lceil \frac{dx(y - y_0) - 1}{dy} \right\rceil + 1 + x_0 \quad (4)$$

If our initial y value is y_0, it's easy to see the initial value in the floor is -1/dy. The floor of this is -1, and $-1 + 1 + x_0 = x_0$, as we expect. We'll be doing *forward differences* to step our edges, so after we generate the initial value for x, we're going to want to step y by 1 to the next scanline and generate the next x from our previous x value, without recalculating it from scratch. I will assume you are already familiar with forward differences, which are covered in any decent computer graphics book,

so I'm just going to point out the interesting parts of this algorithm.

Perhaps the most interesting thing about this particular equation is how the floor interacts with the forward differences, especially when dx is less than 0.

Mod Squad

To thoroughly analyze Equation 4's behavior, we need another trick for manipulating floors:

$$\left\lfloor \frac{a}{b} \right\rfloor = \frac{a}{b} - \frac{a \bmod b}{b} \qquad (5a)$$

You're probably familiar with the modulus operator, `mod`, from programming in C or other languages (in C, % is the mod operator). As long as two numbers, a and b, are positive, a mod b is the integer remainder after dividing the numerator a by the denominator b. Equation 5a says we take the real number a/b and subtract its remainder over b and to get the floored value, an integer.

The mathematically defined mod usually behaves differently, in subtle ways, than the mod in your programming language of choice, and because we're using the "math-mod" in the definition of our fill convention we need to make sure we don't let an ill defined programming language muck up the works. For example, ANSI C (and C++) defines the mod operator to be the same as the math-mod operator when both operands are positive, but when either operand is negative the result is implementation dependent—the standard only defines the relationship of a/b and a%b, not their values, in this case. Fortunately our denominator, dy, is always positive because we step down the polygon from top to bottom, so we only have to deal with the case where the numerator, dx, is negative.

We saw how the floor function behaved with negative values, so if Equation 5a is true (it is, trust me), that dictates how the math-mod behaves as well. Assuming b is positive (our dy), a little thought and some scratch paper will show you that a mod b is always positive regardless of whether a is positive or negative. This is because the floor of a negative number goes to the next lowest number, so the mod term must be positive to bring it back up to the real value of a/b. Figure 3 shows a graph of x mod 3. Here's Equation 5a rearranged to make that more clear:

$$\frac{a}{b} = \left\lfloor \frac{a}{b} \right\rfloor + \frac{a \bmod b}{b} \qquad (5b)$$

Equation 5b shows a fraction as we sometimes think of it with an integer part and a fractional part, since a mod b is always smaller than b.

Even if we want to ignore the ANSI standard and hope our platform calculates mod correctly, we're out of luck on most machines, including Intel x86 processors. The x86 signed divide instruction, idiv, truncates towards 0 when dividing negative numerators, which is exactly the opposite of the real floor function. It appears we need to develop a flooring divide and mod function that works on any standard platform, that is, any platform that computes positive mods and divides correctly.

If a ≥ 0, then we'll just do the normal divide and mod. On the other hand, if a < 0, let m = (-a) mod b:

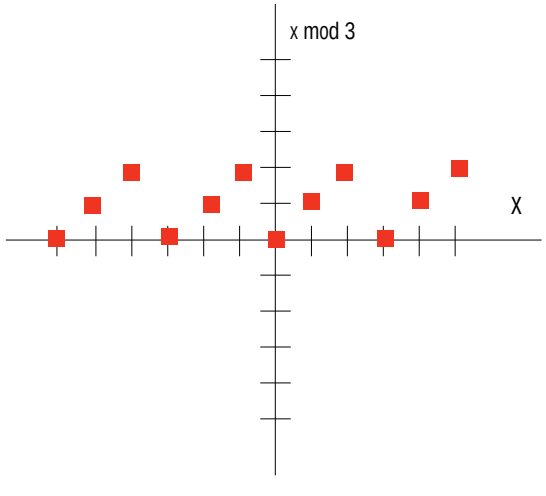
$$\left\lfloor \frac{a}{b} \right\rfloor = \begin{cases} -\left\lfloor \frac{(-a)}{b} \right\rfloor, & m = 0 \\ -\left\lfloor \frac{(-a)}{b} \right\rfloor - 1, & m \neq 0 \end{cases} \qquad (6)$$

$$a \bmod b = \begin{cases} 0, & m = 0 \\ b - ((-a) \bmod b), & m \neq 0 \end{cases} \qquad (7)$$

In other words, Equations 6 and 7 say that if m = 0 (there is no remainder), then we do the flooring divide and mod differently than if there is a remainder. This probably seems really complicated, but if you sit down with a piece of paper and refer to the equations and Figure 3 you'll see how this works in no time (okay, maybe five or ten minutes...it took me a while, too). Our C++ function to correctly compute flooring divides and mods looks like this:

```
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,
```

Figure 3. x mod 3



```
   do the right
 thing
    Floor = -((-Numerator) / Denomina-
tor);
   Mod = (-Numerator) % Denominator;
   if(Mod) {
// there is a remainder
    Floor--; Mod = Denominator - Mod;
   }
  }
}
```

Why?

Let's take a step back and ask ourselves (as you're probably already asking yourself), "Why do we care?" People have been rasterizing polygons since shortly after the beginning of time, and they never went through all this, you say. Well, if their polygons don't have dropouts and consistently light the correct pixels, then they went through all this or its equivalent for another fill convention.

The vast majority of rasterizers don't work properly, and that's why the vast majority of games have dropouts and overwrites at abutting polygon edges. We're taking the time up front to get the math exactly right, so we can implement our rasterizer with total confidence that it will light *exactly* the right pixels; no more, no less.

This is my personal crusade to eliminate dropouts and poor quality rasterizers everywhere, and I'm hoping you'll help me accomplish it. The best part about doing it right is it looks better and isn't any slower at run time than doing it incorrectly, there's just more to understand beforehand.

Vive La Différence

Now that we've got an algorithm for the correct divide and mod on any platform, we can go back to our original goal, which was to implement our fill convention with integer forward differences. We can use Equation 5b to manipulate Equation 4. Let n = dx(y - y_0) - 1:

$$x_{\text{int}} = \left\lfloor \left\lfloor \frac{n}{dy} \right\rfloor + \frac{n \bmod dy}{dy} \right\rfloor + 1 + x_0$$

and

$$x_{\text{int}} = \left\lfloor \frac{n}{dy} \right\rfloor + \left\lfloor \frac{n \bmod dy}{dy} \right\rfloor + 1 + x_0$$

(We can take the floor of n/dy out of the enclosing floor because it's an integer; see Equation 3.)

This is our initial state. We calculate n from our starting y value, do the flooring divide and mod (with our correct algorithm if n is negative), and use the n mod dy term's numerator as our initial error term for our forward difference. (We don't actually do the divide. It's implicit in the way the DDA functions.) Since n mod dy is positive and less than dy, we know that the floor of the n mod dy term is 0 and doesn't affect the initial x. As y steps by 1, our floor term steps by dx/dy (calculated by substituting y = y + 1 in our original equation). Our new x (call it x'), is calculated from:

$$x'_{\text{int}} = x_{\text{int}} + \left\lfloor \frac{n \bmod dy}{dy} + \frac{dx}{dy} \right\rfloor$$

We use Equation 5b on the dx/dy step to get:

$$x'_{\text{int}} = x_{\text{int}} + \left\lfloor \frac{dx}{dy} \right\rfloor + \left\lfloor \frac{n \bmod dy}{dy} + \frac{dx \bmod dy}{dy} \right\rfloor \quad (8)$$

Equation 8 says that as y steps by 1, x steps by the floor of dx/dy, and our error term steps by dx mod dy. Note that mod is always positive, so when our error term numerator exceeds our denominator, dy, we add 1 to the resulting x regardless if we're stepping left or right. This probably differs from other DDAs you've used before—the mathematically defined floor and mod terms work out so that you're always adding 1 when your error term rolls over, not just when you're stepping in the positive direction.

Look Before You Jump

Those of you who have written fixed-point edge rasterizers instead of error-term DDAs are probably wondering why we're going to the trouble of doing a DDA, with its accompanying jumps when the error term rolls over. Even though the jump is in the scanline loop, not the pixel loop, jumps are getting more and more expensive as processors get deeper and deeper pipelines. In fact, on more recent Intel architectures the jump prediction logic makes mispredicted jumps that fall through even more expensive than jumps that are taken on earlier processors. Fear not, there is a good reason to use an error-term DDA instead of fixed-point to scan our edges.

Remember the following lines from our floating-point texture mapper

On recent Intel architectures, jump prediction logic makes mispredicted jumps that fall through more expensive than jumps on earlier processors.

in Part 1:

```
int XStart = ceil(pLeft->X);
float XPrestep = XStart - pLeft->X;
float OneOverZ = pLeft->OneOverZ +
  XPrestep * Gradients.dOneOverZdX;
```

When we start a scanline, we need to step in to the first pixel center from the real edge before we can start drawing, and our interpolants (like 1/z in this snippet) need to step with us. We had to calculate `XPrestep` every scanline, and multiply it by the gradients of all our interpolants to get to the starting pixel center before we could draw. This is because we didn't know how far we were from the first pixel center until we did the `ceil()` call.

Now think about how this works with a DDA. We are stepping from one pixel center to the next directly, and we know exactly how far we had to come from the last pixel center: the floor of dx/dy in x plus 1 in y, or that step plus 1 in x when our error term rolls over (see Equation 8). We never need to calculate our prestep to a pixel center because we're always stepping on pixel centers! Take a minute to think this through—it means we get the advantages of sampling from pixel centers, and we don't pay the prestep multiply. As I've mentioned before, these advantages include rock solid textures that don't swim when you rotate and no "hairy texture" artifacts.

Listing 1 shows the salient parts of the integer rasterizer. Because of space constraints, I've only included the differences from last column's listing. You can pick up the entire listing on CompuServe in the *Game Developer* section of the SD Forum or from ftp://ftp.mfi.com/gdmag/src/.

The code is in a weird state because I left the texture coordinates as floats, while the edge rasterization is in integer coordinates, as we've been discussing. This bizarre combination doesn't affect the rasterizer, and it will be fixed in the next article when we address the texture mapping itself. One thing you may or may not notice when you run this rasterizer is how jerky it is compared to the original floating point rasterizer. If you

Listing 1. The Integer Rasterizer (Continued on p. 26)

```
struct edge {
    edge(gradients const &Gradients, POINT3D const *pVertices,
            int Top, int Bottom );
    inline int Step( void );

    long X, XStep, Numerator, Denominator;        // DDA info for x
    long ErrorTerm;
    int Y, Height;                                // current y and vertical count
    float OneOverZ, OneOverZStep, OneOverZStepExtra;// 1/z and step
    float UOverZ, UOverZStep, UOverZStepExtra;    // u/z and step
    float VOverZ, VOverZStep, VOverZStepExtra;    // v/z and step
};

inline int edge::Step( void ) {
    X += XStep; Y++; Height--;
    UOverZ += UOverZStep; VOverZ += VOverZStep;
    OneOverZ += OneOverZStep;

    ErrorTerm += Numerator;
    if(ErrorTerm >= Denominator) {
        X++;
        ErrorTerm -= Denominator;
        OneOverZ += OneOverZStepExtra;
        UOverZ += UOverZStepExtra; VOverZ += VOverZStepExtra;
    }
    return Height;
}

void DrawScanLine( BITMAPINFO const *pDestInfo, BYTE *pDestBits,
    gradients const &Gradients, edge *pLeft, edge *pRight,
    BITMAPINFO const *pTextureInfo, BYTE *pTextureBits );

/******** TextureMapTriangle **********/

/********** handle floor divides and mods correctly ***********/

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;
        }
    }
}

/********** edge constructor ***********/

edge::edge( gradients const &Gradients, POINT3D const *pVertices.
        int Top, int Bottom )
{
    Y = pVertices[Top].Y;
    Height = pVertices[Bottom].Y - Y;
    int Width = pVertices[Bottom].X - pVertices[Top].X;

    if(Height) {
        // this isn´t necessary because we always start at TopY,
        // but if you want to start somewhere else you´d make
        // Y your start
        FloorDivMod(Width * (Y - pVertices[Top].Y) - 1,
```

compile two test programs, one with each rasterizer, and run them side by side, you'll easily see the quality difference.

The texture mapping is jerky because we use the endpoints of the triangle to compute the gradients, and the endpoints are changing by relatively large amounts as the triangle moves because of the integer truncation. You also see similar jerkiness in a lot of game rasterizers, and it's probably caused by the same thing (compounded with the artifacts generated by not stepping on pixel centers). Even in the low 320 by 200 resolution game world, this jitter is visible separately from the normal aliasing. In accordance with our quest to increase rasterization quality around the world, I find this unacceptable. The solution happens to be simple: fractional endpoints. Unfortunately, I was out of space a while back, and my editor is beginning to hate me, so the description of this solution will have to wait until next time.

Summing Up

Once again, I'm over my word budget, and I still haven't covered everything. I simply must give credit where credit is due, however—without my friend Kirk Olynyk's help and tutelage I'd still be lighting the wrong pixels without knowing the difference. If you're into this kind of discrete math (it's so useful for raster graphics) and you want to learn more, *Concrete Mathematics* (Addison Wesley, 1994) by Ronald L. Graham and Oren Patashnik is great.

Also, while discussing my article "Changing the Rules for Transparent Blts" (Under the Hood, Feb. 1995) on rec.games.programmer, Rich Gortatowsky (rg@raster.kodak.com) mentioned that for best results, your RLE compressor should try to compress vertically as well as horizontally. I totally agree.

Finally, I promise we'll get back to the actual texture mapping portion of the texture mapper next time. ■

Chris Hecker wants a single-cycle integer multiply on future x86 processors so bad he can taste it. Yum yum. You can contact him via e-mail at checker@bix.com or through Game Developer *magazine.*

Listing 1. The Integer Rasterizer (Continued from p. 24)

```
                Height,X,ErrorTerm);
        X += pVertices[Top].X + 1;

        FloorDivMod(Width,Height,XStep,Numerator);
        Denominator = Height;

        OneOverZ = Gradients.aOneOverZ[Top];
        OneOverZStep = XStep * Gradients.dOneOverZdX
                    + Gradients.dOneOverZdY;
        OneOverZStepExtra = Gradients.dOneOverZdX;

        UOverZ = Gradients.aUOverZ[Top];
        UOverZStep = XStep * Gradients.dUOverZdX
                    + Gradients.dUOverZdY;
        UOverZStepExtra = Gradients.dUOverZdX;

        VOverZ = Gradients.aVOverZ[Top];
        VOverZStep = XStep * Gradients.dVOverZdX
                    + Gradients.dVOverZdY;
        VOverZStepExtra = Gradients.dVOverZdX;
    }
}

/********** DrawScanLine ************/

void DrawScanLine( BITMAPINFO const *pDestInfo, BYTE *pDestBits,
    gradients const &Gradients, edge *pLeft, edge *pRight,
    BITMAPINFO const *pTextureInfo, BYTE *pTextureBits )
{
    // assume dest and texture are top-down
    assert((pDestInfo->bmiHeader.biHeight < 0) &&
            (pTextureInfo->bmiHeader.biHeight < 0));

    int DestWidthBytes = (pDestInfo->bmiHeader.biWidth + 3) & ~3;
    int TextureWidthBytes = (pTextureInfo->bmiHeader.biWidth + 3) & ~3;

    int XStart = pLeft->X;
    int Width = pRight->X - XStart;

    pDestBits += pLeft->Y * DestWidthBytes + XStart;

    float OneOverZ = pLeft->OneOverZ;
    float UOverZ = pLeft->UOverZ;
    float VOverZ = pLeft->VOverZ;

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

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

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