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.

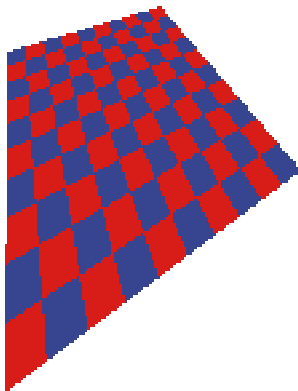
Thank you, and I hope you enjoy the article,

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 I: Foundations

Figure 1. Textured Checkerboard



If there is one technical feature today's high-performance three-dimensional games must have, it is texture mapping. The technique of texture mapping stretches across almost every genre of game, from role-playing games like Ultima Underworld and System Shock, through simulators like Indy Car Racing and Wing Commander III, to action games like Doom and Descent.

Given its popularity, you'd think there would be a wealth of information available on how to actually write your own perspective texture mapper. You'd be wrong.

When I was researching this article (actually, when I was trying to figure out if an article on perspective texture mapping was even needed), I looked high and low for intuitive descriptions and working sample code, but not much exists. Most articles on the Internet describe affine texture mapping, and the few perspective texture mapping articles I did find on x2ftp.oulu.fi, an excellent game programming ftp site managed by Jouni Miettunen, use overly complicated descriptions and aren't accompanied by working code. Even old standbys, like *Computer Graphics: Principles and Practice* (Addison-Wesley, 1992) by James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes (commonly called Foley and van Dam, much to the chagrin of Feiner and Hughes, I'm sure) and the bible of texture mapping, *Digital Image Warping* (1990, IEEE Computer Society) by George Wolberg, are woefully inadequate if you actually want to write a texture mapper, especially one fast enough to be compelling for games.

I'm going to address this lack of documentation in this and the second part of this article. First, using nothing more than basic algebra and geometry, I'll show you an easy-to-understand mathematical foundation, for how and why perspective texture mapping works. I'll also provide sample code to implement the naive algorithm. In the second installment, we'll speed it up to interactive performance.

Assumptions, Definitions, and Concepts

If we want to cover everything in two articles, we're going to have to move pretty fast. To do this, I need to assume you know a bit about three-dimensional graphics. If you don't know how object space, world space, view space, and screen space interact, or you don't know what those terms mean, you should probably pick up a book like Foley and van Dam before reading this article.

The term "texture mapping" describes a whole family of techniques, but for these articles, we'll define texture mapping as drawing a planar polygon as if a bitmap was glued to the polygon's face. This bitmap goes through the same transforms (or at least looks like it does) as the polygon, so if we view the polygon almost edge on, the bitmap, or texture, will look like it's edge on as well. Figure 1 shows a checkerboard viewed at an angle. You can see how the squares get smaller as they recede, just as you'd expect.

To accomplish this mapping, we associate a texture bitmap with each polygon and texture coordinates with each vertex of the polygon. In addition

Chris Hecker

Little has been written on perspective texture mappers, an invaluable feature of any high-performance game. This month, Chris Hecker fills the void with the first of a two-part article on the subject.

to the normal (x,y,z) triplet to define a vertex in three dimensions, we specify the texture coordinates u and v. These coordinates are two-dimensional coordinates into the texture bitmap, and the pixels in this bitmap are sometimes called texels.

To make things easy to visualize, our diagrams and equations will be in two dimensions—think of working in a slice through the three-dimensional space—but our results extend easily into three dimensions.

Perspective Projections

Most three-dimensional game graphics are based on perspective projections. Perspective projections make distant objects seem smaller than closer objects and distort angles so scenes look realistic.

The basic equation for the perspective projection uses similar triangles that share a vertex at the origin (the viewpoint). If we take the point (x_0,z_0) (ignore the u coordinates for the time being) and project it onto the dashed vertical z=d line in Figure 2 to give us (x_0',d), the equation for the relationship between these two points is:

$$\frac{x'_0}{d} = \frac{x_0}{z_0}$$

In other words, the ratio of the height of the triangle formed by ((0,0), (x_0',d), (0,d)) to the length of its base is the same as the ratio of the height of the triangle formed by ((0,0), (x_0,z_0), $(0,z_0)$) to the length of its base. If we assume d=1 for the current example, and generalize this equation to all unprojected points (x,z), we get:

$$x' = \frac{x}{z} \qquad (1)$$

If we view the z=d line as the one-dimensional equivalent of the two-dimensional screen plane (pretend you're looking down on the plane from above, so you can only see it as a line), Equation 1 says we can generate screen coordinates (x´ for values of x) by dividing the unprojected object coordinates by their z values. This is the perspective projection in its essence.

Mapping Direction

In three-dimensional graphics, we consider transforming from object to screen coordinates moving "forward," so Equation 1 is called a forward mapping—it projects the source polygon forward onto destination pixels. To use a forward mapping for texturing a polygon, you step along the polygon in object space and project each generated point forward to a destination pixel position. Forward mappings don't work very well for texture mapping, however, because it's hard to be sure how far to step in the source so that the projected coordinates don't skip or overwrite any pixels in the destination.

Backward mappings, on the other hand, allow us to step in screen space, processing each pixel exactly once. If we manipulate Equation 1 to give us a backward mapping from x´ to x we get:

$$x = x'z \qquad (2)$$

This tells us we can generate values of x from values of x´ if we multiply x´ by z. We can easily generate the desired u texture coordinate once we have x, but first we must find the correct z to feed into

Equation 2 (we already know x´ because it's the current pixel we want to write).

It would be great if we could generate z values directly from x´ values using a simple linear interpolation. We often use linear interpolations in graphics in the form of digital differential analyzers (DDAs), and fixed and floating-point interpolations, but we know linear interpolation is only accurate when we are interpolating a linear equation. Let's explore the relation between x´ and z to see whether they are linear with respect to one another, which in turn will tell us if we can use linear interpolation to generate z from x´.

A linear equation is any equation of the form:

$$y = AX + B \tag{3}$$

for any real values of A and B (this is called the slope-intercept form, where A is the slope of the x,y line, and B is the y-intercept, or value of y when the line crosses the y axis). That is, as x changes by a constant amount, y changes by a constant amount proportional to the change in x.

To find the relationship between x´ and z, we first take the equation for the unprojected line in object space, x = Az + B. The actual values of the constants A and B are based on the endpoints of the line segment, and are irrelevant to this derivation. Next, we substitute this into Equation 2 to get an equation in z and x´, and solve for z:

$$Az + B = x'z$$
$$B = z(x'-A)$$

and finally:

$$z = \frac{B}{x' - A} \tag{4}$$

Equation 4 is definitely not a linear equation with respect to x´, so we can't directly compute z incrementally from values of x´. However all is not lost, because a little algebraic manipulation gives us:

$$\frac{1}{z} = \frac{1}{B}x' - \frac{A}{B} \tag{5}$$

Equation 5 is a linear equation with respect to x´. The only problem is, it's a linear equation of 1/z with respect to x´, not z itself! We can use Equation 5 to linearly interpolate values of 1/z and take the reciprocal at each pixel to get the real value of z. In other words, we can linearly interpolate 1/z and divide x´ by 1/z to generate values of x according to Equation 2. These values of x allow us to compute values of u that we can use to look up the correct color from the texture bitmap to store in x´. Voila, perspective texture mapping.

It turns out we can compute u directly instead of computing x, saving a step and simplifying our lives. By definition in Equation 1, x/z is linear in screen space (it's actually equal to screen space, which is about as linear as you can get!). Just as x and z are linear with respect to each other because the object is planar (or linear, in Figure 2), u and x are linear with respect to each other for the same reason. Well, if x/z is linear in screen space, and x is linear with u, then u/z is linear in screen space as well (you can prove this to yourself by playing around with Equations 1, 3, and 5). Instead of dividing x/z by 1/z to generate x coordinates that we then use to solve for u coordinates, we can interpolate u/z and divide it by 1/z to generate the u values directly.

Affine texture mapping ignores these results and linearly interpolates u and v in screen space without the divide. This results in funky warping, but for some polygons it's not too bad (and because there's no divide it has the potential to be a lot faster). A comparison is beyond the scope of this article, but you can find affine texture mappers on x2ftp.oulu.fi, which I mentioned previously.

Our Story So Far

Let's take a break, sum up our results to this point, and outline a simple algorithm to perspective texture map the line segment in Figure 2.

We've shown that 1/z and u/z are linear in screen space, so the algorithm for texture mapping Figure 2 goes like this:

- Project the object vertices into screen space, giving x´ = x/z and u´ = u/z.
- Let z´ = 1/z at each vertex.
- Linearly interpolate u´ and z´ between x_0' and x_1', stepping x´ by 1 pixel each loop.
- At each pixel x´, calculate u by u´/z´, and use u to fetch the correct texel.
- Write the texel to the destination at x´.

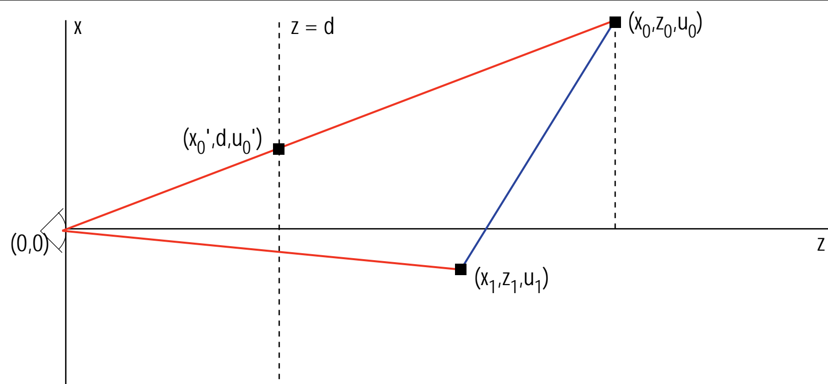
The proofs for y and v are analogous to those for x and u, so this is all there is to writing a three-dimensional perspective texture mapper.

Interpolation Breakdown

In the simplified algorithm I've outlined, we linearly interpolated u´ and z´ over the length of the scanline. Each linear interpolation usually involves these steps:

- Figure out the start and end values of the interpolants (in Figure 2, u_0',z_0' and u_1',z_1', respectively).

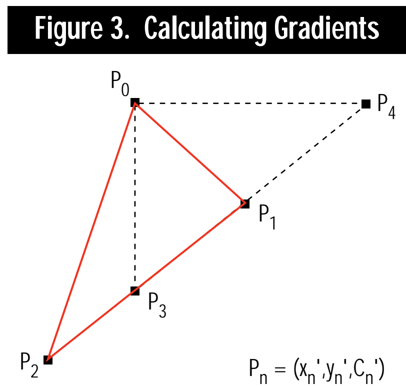
Figure 2. Perspective Projection



- Calculate the amount each changes as it moves from start to end ($u_1' - u_0'$ and $z_1' - z_0'$).
- Divide the change by the distance over which you want to interpolate ($x_1' - x_0'$) to get each step.
- Increment from the start to the end by this step.

This is a fair amount of work, and if we plan to rasterize polygons like the triangle shown in Figure 3, we have even more work to do. We need to interpolate at least 1/z, u/z, and v/z (and possibly one or three colors), and if we first calculate the interpolants down each edge, and then calculate new ones when we get to each scanline we will soon get lost in a sea of interpolants going in all sorts of directions. Luckily, there is a better way.

Figure 3. Calculating Gradients



It just so happens that the increments in each linear interpolant for a single step in x or in y are constant across the whole polygon. This is a very important and very cool result because it means we can calculate these increments—called the gradients—once and never need to worry about calculating interpolants again during rasterization. In other words, when we want to rasterize a polygon, we calculate the gradients in x and in y for each parameter at the very beginning, and every time we step in x or y or both we just add in the appropriate gradients. When we get to a scanline we want to draw, we don't need to calculate linear interpolations for all of our parameters as we step across the scanline in x, because we already have their gradients with respect to x sitting around! In the same vein, stepping down an edge is simply some combination of the gradient in x and the gradient in y. (If you think about it, this also means you only need to interpolate the parameters down one edge. Ponder that one for a while.)

To show how gradients are calculated, let's use the triangle $P_0P_1P_2$ in Figure 3. Each vertex has a screen space x and y associated with it (x', y'), but in addition there is an arbitrary parameter, c', which could be color for Gouraud shading or 1/z, u/z, or v/z for perspective texture mapping. It is any parameter we can linearly interpolate over the surface of the two-dimensional (screen space) triangle.

Given this triangle, let's figure out how the parameter c' changes if we hold y constant and step in x. We will use the point P_4 in our construction (P_3 and P_4 are both on the P_1P_2 line in Figure 3). It is clear that $y_4' = y_0'$, and we can derive the other coordinates for P_4 using the line equations:

$$\frac{x_1' - x_2'}{y_1' - y_2'} = \frac{x_4' - x_2'}{y_4' - y_2'}$$

and:

$$\frac{c_1' - c_2'}{y_1' - y_2'} = \frac{c_4' - c_2'}{y_4' - y_2'}$$

Substituting y_0' for y_4' and solving for the various coordinates gives us:

$$y_4' = y_0'$$

$$x_4' = \left(\frac{x_1' - x_2'}{y_1' - y_2'}\right)(y_0' - y_2') + x_2'$$

and:

$$c_4' = \left(\frac{c_1' - c_2'}{y_1' - y_2'}\right)(y_0' - y_2') + c_2'$$

Next, refer to Figure 5 to compute the difference in c' (called dc') as it moves from P_0 to P_4 with Equation 6.

The analog for c' with respect to y as we move from P_0 to P_3 is also shown in Figure 5, in Equation 7. (Notice the denominators: dx = -dy.)

The values dc'/dx and dc'/dy are called the gradients for the parameter; dc'/dx is the gradient with respect to x and dc'/dy is the gradient with respect to y. We can calculate the gradients for 1/z, u/z, and v/z with respect to both x and y at the top of our texture mapper and never need to calculate them again during the rasterization of this polygon.

Our new texture mapping algorithm looks like this:

- Project the object vertices into screen space, giving x', y', $u' = u/z$, $v' = v/z$, and $z' = 1/z$.
- Calculate the gradients in x and y for u', v', and z'.
- Linearly interpolate down each edge and across each scanline using the gradients.
- At each pixel, calculate u by u'/z' and v by v'/z', and use u and v to fetch the correct texel.
- Write the texel to the destination at x', y'.

The only thing we're missing is a consistent fill convention to make sure we light the correct destination pixels as we rasterize the polygon. Once we have a fill convention, we can guarantee polygons will abut properly and we won't have any skipped pixels (dropouts) or overwrites at the edges.

Conventional Wisdom

A fill convention is a set of rules that describes how to light pixels in the screen under various edge conditions. The first step towards implementing a fill convention is defining exactly which pixels we want lit when a polygon is rasterized. Figure 4 shows the raster grid of the display, with pixel centers marked with black dots.

We will define what's called a top-left fill convention. Top-left refers to the tie breaking rule used when the edge of a polygon lands exactly on a pixel center; if the edge is a top or a left edge, the pixel is in the polygon, if it's a right or a bottom edge, the pixel is considered out. You can see this convention in Figure 4. If the red edge is shared by the blue and the yellow polygon, they will not light any of the same pixels. The horizontal red edge is the top of the yellow polygon, so the pixels are considered members of that polygon. In contrast, the horizontal red edge is a bottom edge of the blue polygon, so the pixels are not lit. All

other pixels—those not intersected on pixel centers—are lit if they are "strictly in" the polygon. In other words, the pixel center must be completely inside the edge for the pixel to be lit. In contrast with Figure 4, real edges are infinitely thin, so the pixel center is either out, intersected exactly, or in.

The next step is to define the fill convention mathematically. A top-left fill convention is defined by the ceiling function for the left and top edges, and the ceiling-1 of the right and bottom edges. (The ceiling function bumps a fractional number up to the next integer unless it's already an integer, in which case the number stays the same.) We'll be stepping in y to generate scanlines, so the equation for generating x coordinates from y coordinates as we step from P_0 to P_2 in Figure 3 is:

$$x = \left(\frac{x_2' - x_0'}{y_2' - y_0'} \right) (y - y_0') + x_0'$$

We apply the ceiling function to this equation to give us integer raster values for a given y:

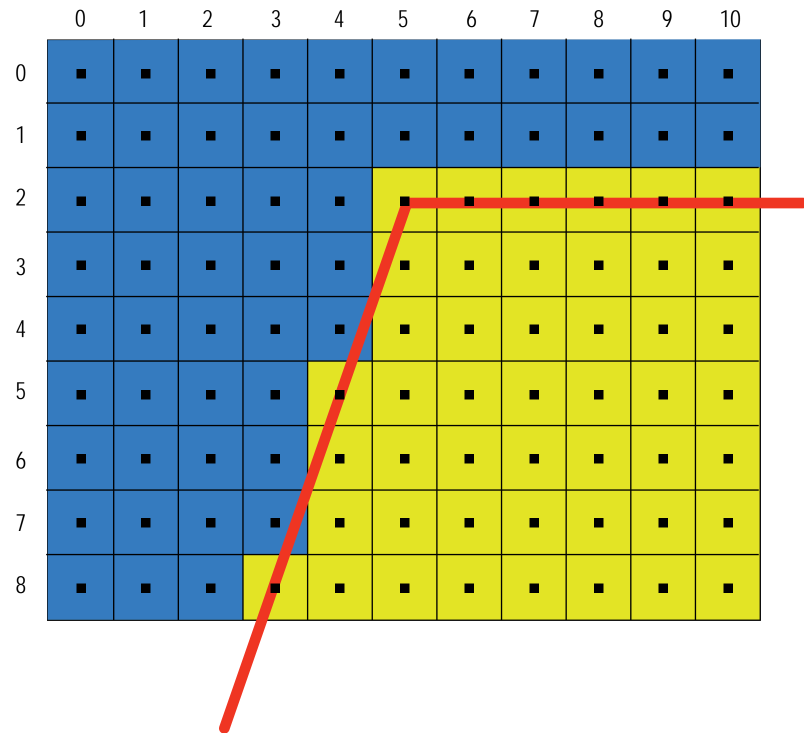
$$x_{\text{int}} = \left\lceil \left(\frac{x_2' - x_0'}{y_2' - y_0'} \right) (y - y_0') + x_0' \right\rceil$$

If our starting coordinates are real numbers instead of integers, we need to apply our convention to the y coordinate as well to generate the initial y value:

$$y_{0\,\text{int}}' = \left[y_0' \right]$$

On a number of scanlines in Figure 4, the real edge—the line on which we're interpolating our parameters—differs from the starting pixel center by some small amount. Pixels in the display are not points, they're actually boxes with an area around the pixel center (the pixel center is the integer coordinate, and the box extends 0.5 pixels to each side), and when stepping from pixel to pixel we want to make sure we step from one pixel center to the next. If we don't step on pixel centers, our textures will appear to swim as our polygon rotates because we aren't sampling from the same place in the pixel each time. Also, when reading from what are essentially random places in each pixel it's possible to generate texture coordinates outside the texture bitmap (which could crash our program).

Figure 4. Fill Conventions



Find Your Center

Given that we want to sample the texture from the exact pixel center, we need to make sure our interpolants are prestepped on each scanline by the difference between the real edge and x_{int}. If we do this correctly, our texture mapper will never read outside the texture (assuming the texture coordinates are valid in the first place, of course), and our textures will not swim as our polygon moves around the screen. Also, we won't get the "hairy texture" artifacts you see in a rasterizer that doesn't step on pixel centers, where lines in the texture that should be straight come out with little notches and pimples.

Figure 6 shows a close-up of a group of pixels. To start rasterizing, we must first step our edge to the point A. This involves an x and y prestep for our interpolants, marked with dotted lines. Now, we can precalculate each parameter's step in y and in x for a single scanline step in the screen using the gradients we calculated beforehand, so each time we move from one scan to the next, we just add each step to its interpolant to find the new value. When it's time to draw a scanline, we must step to the first pixel center. Figure 6 shows this step as a dotted line at each scanline.

Figure 5. Equations 6 and 7

$$\frac{dc'}{dx} = \frac{c_4' - c_0'}{x_4' - x_0'} = \frac{(c_1' - c_2')(y_0' - y_2') - (c_0' - c_2')(y_1' - y_2')}{(x_1' - x_2')(y_0' - y_2') - (x_0' - x_2')(y_1' - y_2')} \quad (6)$$

$$\frac{dc'}{dy} = \frac{c_3' - c_0'}{y_3' - y_0'} = \frac{(c_1' - c_2')(x_0' - x_2') - (c_0' - c_2')(x_1' - x_2')}{(x_0' - x_2')(y_1' - y_2') - (x_1' - x_2')(y_0' - y_2')} \quad (7)$$

All this prestepping probably sounds expensive, but there is a way to do it that requires no extra multiplies per scanline. We'll discuss this in more detail next month.

Summary and Random Notes

This article is too long already, but there's still plenty we haven't discussed.

First, we didn't talk about all the special cases where linearly interpolating the texture coordinates actually is correct, like walls and floors. A close examination of the math above will show you why this is true (hint: look at the gradients for the 1/z term). Games like Doom use this to speed up their texture mappers at the expense of not allowing arbitrarily oriented polygons. There's lots of information covering these techniques on the Internet.

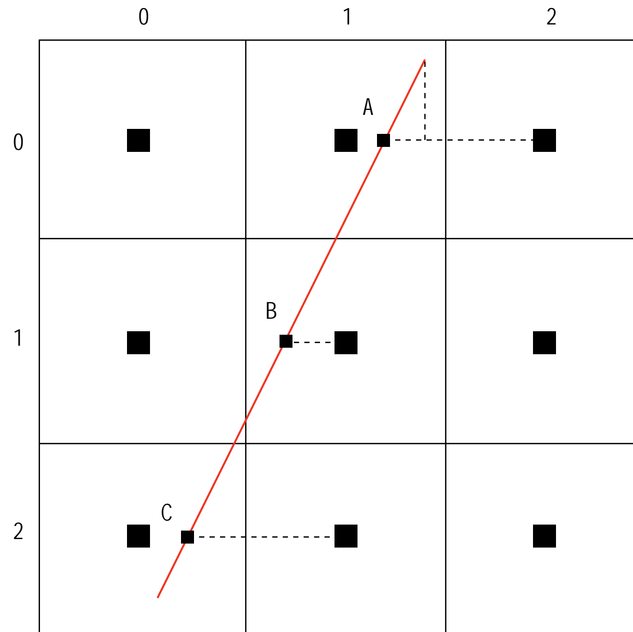
We also didn't discuss antialiasing or homogeneous coordinate systems. *Digital Image Warping* is a great resource for antialiasing and image resampling, while Foley and van Dam cover homogeneous coordinates.

Even considering what we missed, we certainly covered a lot of material in a small space, and I encourage you to reread this article with a piece of paper in hand and try to prove the various results for yourself.

The sample code included with this article implements the perspective texture mapping algorithm. It is a high quality implementation with one small problem: it's a bit slow, doing a divide and two multiplies per pixel. In the next column I'll show how to optimize this code, which will give you a production quality perspective texture mapper you can just plop right in your game engine. ■

Chris Hecker wishes he had a Ph.D. in mathematics so he didn't have to struggle with the derivation of the equation for the area of a triangle whenever he wanted to use it. In the meantime, he can be reached at checker@bix.com or through Game Developer *magazine.*

Figure 6. Pixel Centers



Listing 1. Perspective Texture Mapper

```
#include<windows.h>
#include<math.h>

struct POINT3D {
    float X, Y, Z;
    float U, V;
};

void TextureMapTriangle( BITMAPINFO const *pDestInfo,
        BYTE *pDestBits, POINT3D const *pVertices,
        BITMAPINFO const *pTextureInfo,
        BYTE *pTextureBits );

/******** structures, inlines, and function declarations **********/

struct gradients {
    gradients( POINT3D const *pVertices );
    float aOneOverZ[3];             // 1/z for each vertex
    float aUOverZ[3];               // u/z for each vertex
    float aVOverZ[3];               // v/z for each vertex
    float dOneOverZdX, dOneOverZdY; // d(1/z)/dX, d(1/z)/dY
    float dUOverZdX, dUOverZdY;     // d(u/z)/dX, d(u/z)/dY
    float dVOverZdX, dVOverZdY;     // d(v/z)/dX, d(v/z)/dY
};

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

    float X, XStep;                 // fractional x and dX/dY
    int Y, Height;                  // current y and vert count
    float OneOverZ, OneOverZStep;   // 1/z and step
    float UOverZ, UOverZStep;       // u/z and step
    float VOverZ, VOverZStep;       // v/z and step
};

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

Listing 1. Perspective Texture Mapper (Continued on p. 25)

```
            pLeft,pRight,pTextureInfo,pTextureBits);
        TopToMiddle.Step(); TopToBottom.Step();
    }

    Height = MiddleToBottom.Height;

    if(MiddleIsLeft) {
        pLeft = &MiddleToBottom; pRight = &TopToBottom;
    } else {
        pLeft = &TopToBottom; pRight = &MiddleToBottom;
    }

    while(Height--) {
        DrawScanLine(pDestInfo,pDestBits,Gradients,
            pLeft,pRight,pTextureInfo,pTextureBits);
        MiddleToBottom.Step(); TopToBottom.Step();
    }
}

/********** gradients constructor **********/

gradients::gradients( POINT3D const *pVertices )
{
    int Counter;

    float OneOverdX = 1 /(((pVertices[1].X - pVertices[2].X) *
            (pVertices[0].Y - pVertices[2].Y)) -
            ((pVertices[0].X - pVertices[2].X) *
            (pVertices[1].Y - pVertices[2].Y)));

    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]) *
            (pVertices[0].Y - pVertices[2].Y)) -
            ((aOneOverZ[0] - aOneOverZ[2]) *
            (pVertices[1].Y - pVertices[2].Y)));
    dOneOverZdY = OneOverdY * (((aOneOverZ[1] - aOneOverZ[2]) *
            (pVertices[0].X - pVertices[2].X)) -
            ((aOneOverZ[0] - aOneOverZ[2]) *
            (pVertices[1].X - pVertices[2].X)));

    dUOverZdX = OneOverdX * (((aUOverZ[1] - aUOverZ[2]) *
            (pVertices[0].Y - pVertices[2].Y)) -
            ((aUOverZ[0] - aUOverZ[2]) *
            (pVertices[1].Y - pVertices[2].Y)));
    dUOverZdY = OneOverdY * (((aUOverZ[1] - aUOverZ[2]) *
            (pVertices[0].X - pVertices[2].X)) -
            ((aUOverZ[0] - aUOverZ[2]) *
            (pVertices[1].X - pVertices[2].X)));

    dVOverZdX = OneOverdX * (((aVOverZ[1] - aVOverZ[2]) *
            (pVertices[0].Y - pVertices[2].Y)) -
            ((aVOverZ[0] - aVOverZ[2]) *
            (pVertices[1].Y - pVertices[2].Y)));
    dVOverZdY = OneOverdY * (((aVOverZ[1] - aVOverZ[2]) *
            (pVertices[0].X - pVertices[2].X)) -
            ((aVOverZ[0] - aVOverZ[2]) *
            (pVertices[1].X - pVertices[2].X)));
}

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

edge::edge( gradients const &Gradients,
        POINT3D const *pVertices, int Top, int Bottom )
{
    Y = ceil(pVertices[Top].Y);
    int YEnd = ceil(pVertices[Bottom].Y);
```

```
    OneOverZ += OneOverZStep;
    return Height;
}

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

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

void TextureMapTriangle( BITMAPINFO const *pDestInfo,
        BYTE *pDestBits, POINT3D const *pVertices,
        BITMAPINFO const *pTextureInfo,
        BYTE *pTextureBits )
{
    int Top, Middle, Bottom;
    int MiddleCompare, BottomCompare;
    float Y0 = pVertices[0].Y;
    float Y1 = pVertices[1].Y;
    float Y2 = pVertices[2].Y;

    // sort vertices in y
    if(Y0 < Y1) {
        if(Y2 < Y0) {
            Top = 2; Middle = 0; Bottom = 1;
            MiddleCompare = 0; BottomCompare = 1;
        } else {
            Top = 0;
            if(Y1 < Y2) {
                Middle = 1; Bottom = 2;
                MiddleCompare = 1; BottomCompare = 2;
            } else {
                Middle = 2; Bottom = 1;
                MiddleCompare = 2; BottomCompare = 1;
            }
        }
    } else {
        if(Y2 < Y1) {
            Top = 2; Middle = 1; Bottom = 0;
            MiddleCompare = 1; BottomCompare = 0;
        } else {
            Top = 1;
            if(Y0 < Y2) {
                Middle = 0; Bottom = 2;
                MiddleCompare = 3; BottomCompare = 2;
            } else {
                Middle = 2; Bottom = 0;
                MiddleCompare = 2; BottomCompare = 3;
            }
        }
    }

    gradients Gradients(pVertices);
    edge TopToBottom(Gradients,pVertices,Top,Bottom);
    edge TopToMiddle(Gradients,pVertices,Top,Middle);
    edge MiddleToBottom(Gradients,pVertices,Middle,Bottom);
    edge *pLeft, *pRight;
    int MiddleIsLeft;

    // the triangle is clockwise, so
    // if bottom > middle then middle is right
    if(BottomCompare > MiddleCompare) {
        MiddleIsLeft = 0;
        pLeft = &TopToBottom; pRight = &TopToMiddle;
    } else {
        MiddleIsLeft = 1;
        pLeft = &TopToMiddle; pRight = &TopToBottom;
    }

    int Height = TopToMiddle.Height;

    while(Height--) {
        DrawScanLine(pDestInfo,pDestBits,Gradients,
```

Listing 1. Continued from p. 22

```
    Height = YEnd - Y;

    float YPrestep = Y - pVertices[Top].Y;

    float RealHeight = pVertices[Bottom].Y - pVertices[Top].Y;
    float RealWidth = pVertices[Bottom].X - pVertices[Top].X;

    X = ((RealWidth * YPrestep)/RealHeight) + pVertices[Top].X;
    XStep = RealWidth/RealHeight;
    float XPrestep = X - pVertices[Top].X;

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

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

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

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

void DrawScanLine( BITMAPINFO const *pDestInfo,
        BYTE *pDestBits, gradients const &Gradients,
        edge *pLeft, edge *pRight,
        BITMAPINFO const *pTextureInfo,
        BYTE *pTextureBits )
{
    // we assume dest and texture are top-down

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

    int XStart = ceil(pLeft->X);
    float XPrestep = XStart - pLeft->X;

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

    int Width = ceil(pRight->X) - XStart;

    float OneOverZ = pLeft->OneOverZ +
            XPrestep * Gradients.dOneOverZdX;
    float UOverZ = pLeft->UOverZ +
            XPrestep * Gradients.dUOverZdX;
    float VOverZ = pLeft->VOverZ +
            XPrestep * Gradients.dVOverZdX;

    if(Width > 0) {
        while(Width--) {
            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;
        }
    }
}
```