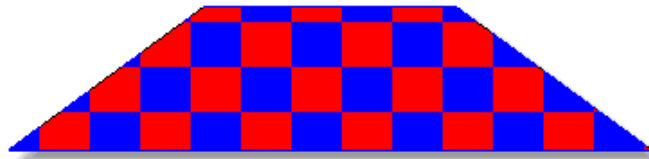


# Texture Mapping Mania

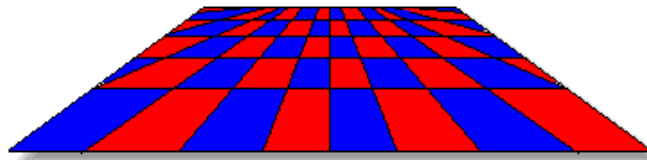
by Andre' LaMothe

A long time ago in a galaxy far, far away, there were only flat shaded 3D games and engines... Today, if your product doesn't support texture mapping then it's probably going to end up in the bargain bin right next to yet another version of *Defender*. Not only is texture mapping expected these days, it's possible in real-time. And it can be accomplished totally in software which makes it nice for programmers. Now the bad news...Texture mapping is actually a very complex subject especially when you start talking about perspective correct, Mip-Mapped, light interpolated, alpha channeled, texture mapping.

Figure 1.0 - Affine and perspective texture mapped polygons.



a. Affine texture mapping - notice no perspective cues.



b. Perspective texture mapping - notice 3D perspective both near and far.

In this article, we are going to cover the most basic form of texture mapping which is referred to as "affine" texture mapping. An affine transformation means that it preserves quantities, thus as one image is mapped onto another and there is a one to one relationship, that is, no warping. In the realm of texture mapping, affine mapping usually means throwing away the 3D information all together and performing a simple 2D mapping. Perspective texture mapping on the other hand, takes the Z coordinate of a 3D polygon definition into consideration hence the perspective warping into consideration that occurs in a 3D space and uses this information to perform the texture mapping in a more visually correct way. Figure 1.0 shows the difference between a affine texture mapped polygon and a perspective correct texture mapped polygon. Notice how the perspective correct image looks more realistic.

The problem with perspective correct texture mapping is that it takes 2 divisions per pixel. Of course, you can approximate perspective correct texture mapping with what is referred to as "perspective correct-ed" texture mapping, by only computing the perspective every few pixels and then affine texture mapping (or

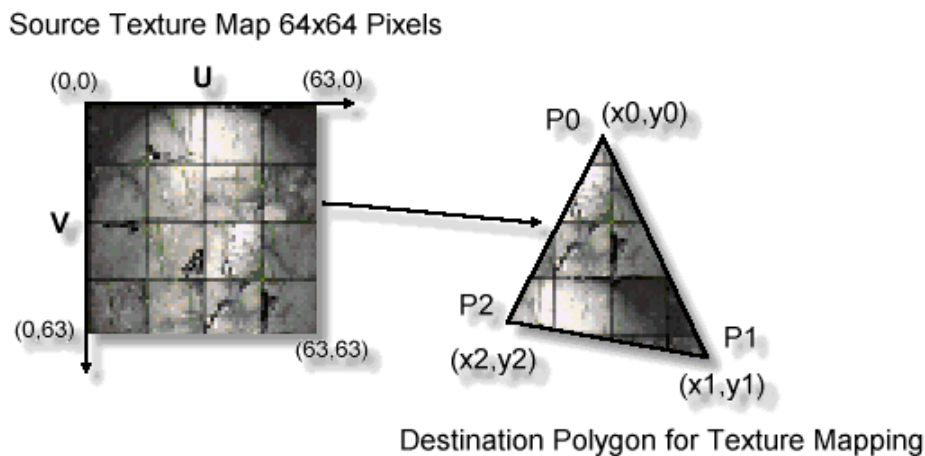
linearly interpolating) in-between. But again this is only an approximation, and you must affine texture map in the middle.

The point is, the ultimate goal is to create a perspective corrected texture mapper with all the toppings. But, you have to start somewhere, and affine texture mapping is a good place since many of the concepts apply to other aspects of 3D rendering such as light interpolation and other sampling type operations. In fact, texture mapping is really nothing more than an application of sampling theory. That's what we're doing, sampling one data set and projecting or mapping it into another. But, enough talk, let's get to it shall we!

## What's In It For Me?

Before we get started on the math and the code, I want to give you an idea of what exactly you're going to get. By the end of this article you'll have all you need including the working source code for an affine texture mapper that can texture map a rectangular bitmap texture that's 64x64 pixels in 256 colors onto a triangular polygon with full texture coordinate support. In addition, I'm going to give you a full working demo that loads in some texture maps and draws thousands of textured triangles a second on the screen. The demo will be in DirectX since the DirectX SDK is readily available and DirectX is the easiest 32 bit platform these days and the one any graphics programmer on the PC should be using. But if you're working on another system then the ideas and concepts are absolutely applicable and the texture mapper is in straight C, so it's totally portable.

Figure 2.0 - Texture Mapping Source to Destination Labeling.



## Getting Down To Specifics

Figure 2.0 shows the exact process that we want to implement. We want to texture map a rectangular bitmap that is 64x64 pixels in 256 colors (1 byte per pixel) onto an arbitrary triangle with any coordinates. This means that we need a way to take rotation and scaling of the triangle into consideration. To help

design the algorithm, I have labeled a number of points of interest on Figure 2.0. First, you'll see that the destination triangle is made up of 3 vertices, these have been labeled  $p_0$ ,  $p_1$ , and  $p_2$ , with coordinates  $(x_0, y_0)$ ,  $(x_1, y_1)$ , and  $(x_2, y_2)$  respectively. In addition, I have labeled the axes around the texture map as U and V, where U is the horizontal axis and V is the vertical axis. Note that both U and V range from (0,0) in the upper left hand corner to (63,63) in the lower right hand corner.

Figure 3.0 - The Different Possible Triangle Types.

Type 1 - Flat Top.      Type 2 - Flat Bottom.      Type 3 - General.



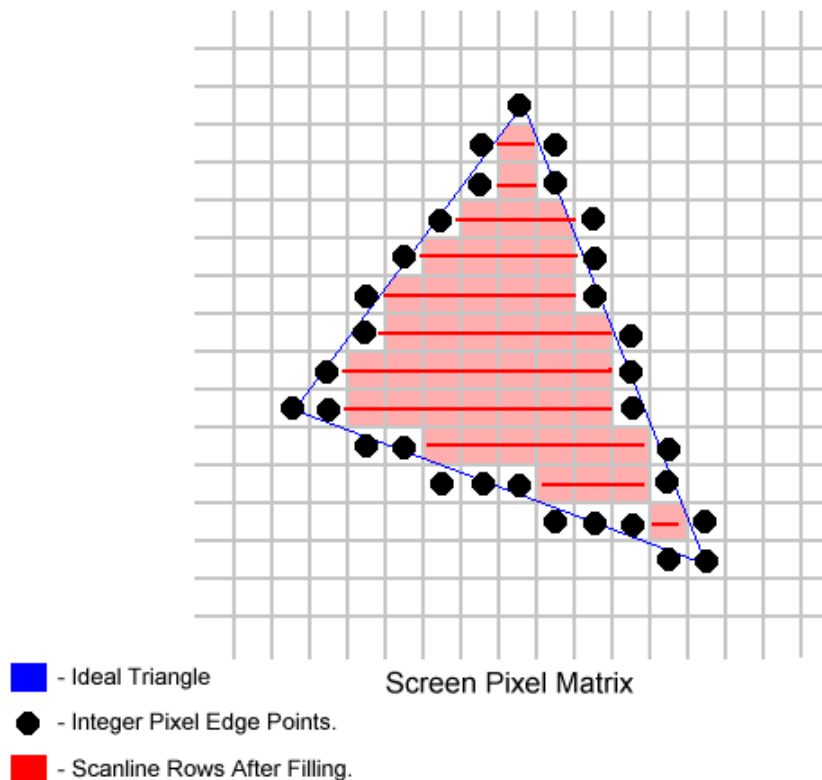
Note: there are really two cases of General depending on which side is longer.

What we want to do is come up with an algorithm that samples the texture map, so that the sampled pixels can be used to color each pixel of each scanline of the target triangle polygon as it is being rendered. Hold on a minute! What's all this about rendering triangles? We'll, the point of texture mapping is to draw or render a triangle with the texture on it, but to draw a textured triangle. I guess we better take a quick look at how to draw an non-textured single color triangle huh? Take a look at Figure 3.0 here you see the various cases of triangles that can possibly exist. There are:

- Type 1: Flat Top      - A triangle with a flat top.
- Type 2: Flat Bottom      - A triangle with a flat top.
- Type 3: General      - A general triangle with neither a flat top or bottom.

However, the general triangle is really made up of a triangle with a flat top, and one with a flat bottom. So if we can draw Type 1 and 2 triangles then Type 3 is easy. So let's focus on Type 2, from it you can figure out how to do Type 1 since it's the same as Type 2, just upside down.

Figure 4.0 - Rasterization of Triangle.



## Interpolate Me Baby!

There are a number of ways to draw triangles including tracing the two edges of the triangle with a line drawing algorithm such as Bresenham's or with simple interpolation. I prefer interpolation since it's more straight forward. Let's see how this works. Take a look at Figure 4.0, all we have to do is find the points that make up the integer rasterized version of the triangle. These points are shown in the figure as little dots. Once we find these dots for each scanline the makes up the triangle then drawing the triangle is nothing more than performing a simple memory fill from dot to dot as shown in Figure 4.0.

Finding these points is nothing more than interpolating the slope (well almost) of each side of the triangle. The interpolation is done as follows:

We know that the height of the triangle is:

$$dy = (y2 - y0) ;$$

And the difference in X's between the lower left vertex and the lower right vertex is:

$$\begin{aligned} dx\_left\_side &= (x2 - x0); \\ dx\_right\_side &= (x1 - x0); \end{aligned}$$

Thus, the slope of the left hand side is:

$$slope\_left\_side = dy/dx\_left\_side = (y2 - y0)/(x2 - x0);$$

And, the slope of the right hand side is:

```
slope_right_side = dy/dx_right_side = (y2 - y0)/(x1 - x0);
```

However, we don't exactly want the slope. The slope is the "change in Y per change in X". This means that if we were to move over exactly one pixel in the X direction then the Y would change by the slope. We don't want this, we want the opposite, or  $dx/dy$ . This is because we are drawing the triangle scan line by scan line and incrementing Y each time, hence  $dy = 1$  which is a constant, thus:

```
dx_left_side = 1 * (x2 - x0)/(y2 - y0);
```

and,

```
dx_right_side = 1 * (x1 - x0)/(y2 - y0);
```

That's it! That's the entire triangle drawing algorithm for a flat bottom Type 2 triangle. Type 1 is similar and I leave up to you. Take a look at Listing 1.0 for a pseudo-code implementation of the triangle drawing algorithm.

#### Listing 1.0 - Pseudo Code Implementation of Triangle Renderer

```
void Draw_Triangle(float x0,float y0,float x1,float y1,float x2,float y2, int color)
{
    // this function rasterizes a triangle with a flat bottom

    // compute left side interpolant
    float dx_left = (x2 - x0)/(y2 - y0);

    // compute right side interpolant
    float dx_right = (x1 - x0)/(y2 - y0);

    // seed left and right hand interpolators
    float x_left = x0;
    float x_right = x0;

    // enter into rasterization loop
    for (int y=y0; y<=y1; y++)
    {
        // draw the scanline
        Draw_Line(x_left, x_right, y, color);

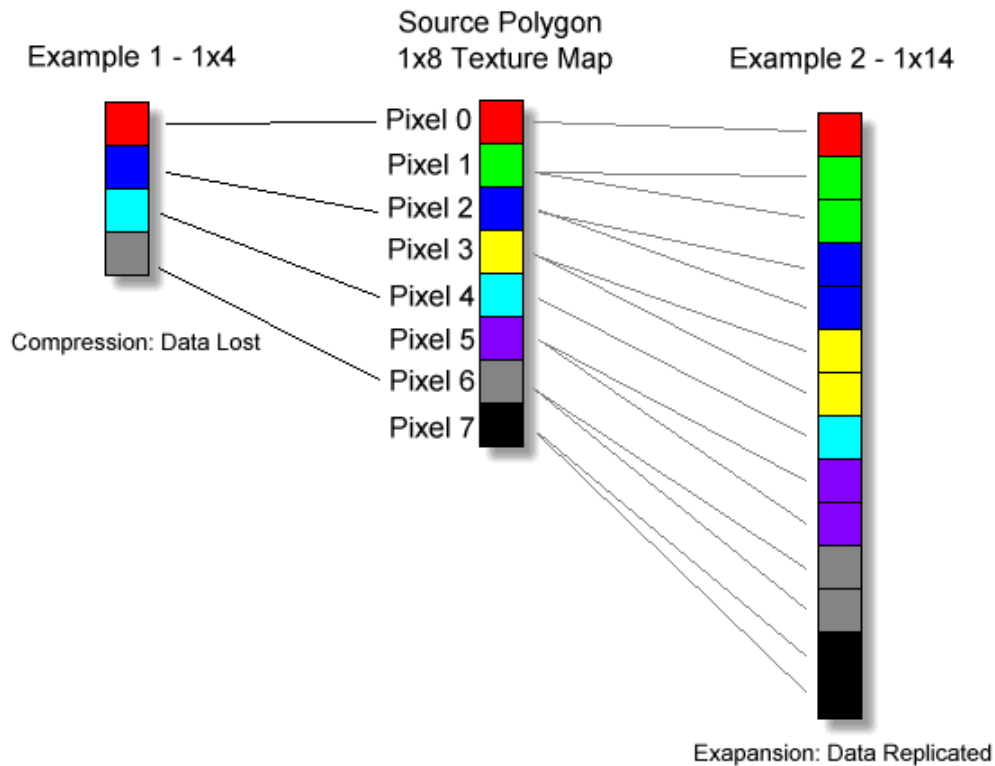
        // advance interpolants
        x_left+=dx_left;
        x_right+=dx_right;

    } // end for y
} // end Draw_Triangle
```

At this point you've seen how interpolation can be used to sample something and draw it. Granted in this case, we didn't actually sample anything more than a single color, but we could of! But the point is that we used the vertical and horizontal deltas for each of the triangle sides to figure out how much to step each vertical scanline step. Hence, the correct X position was what we were interpolating. Get it?

It's very important that you understand the idea of interpolating since the entire texture mapping algorithm is based on it, so take you time and work through the example above with some real numbers to get a better feel for it. I always like to try an algorithm with real numbers for just a few iterations, to get a feel for it. Now let's move onto the using interpolation to texture map a one dimensional polygon and then move on to triangles in any orientation or scale.

Figure 5.0 - 1D Texture Mapping Examples.



## 1-D Interpolations

Texture mapping a triangle with a rectangular texture map is nothing more than a lot of interpolating, but there is so much interpolating that it's easy to make a mistake and or write a slow algorithm, so let's take our time and start with the simplest case in 1 dimension. Figure 5.0 illustrates the worlds simplest texture mapper, the texture mapping of a single vertical line. In Figure 5.0 we have a texture map that is exactly one pixel thick and 8 pixels high, we want to map this into a destination polygon that is exactly one pixel thick, but any height. How do we do this? Again sampling comes to the rescue.

What we need to do is "sample" the texture map which in this case is a single 1x8 pixel bitmap and map it into the destination polygon which is 1xn pixels where n can range from 1 to infinity. Take a look at Figure 5.0 for the derivation of the following examples.

### Example 1

As a first example, let's say that our destination polygon is 1x4 pixels. It makes sense that we want to sample the source texture every other pixel, as shown in the figure. Thus, if we select pixels (0,2,4,6) of the source texture and map them into the destination polygon at positions (0,1,2,3) then we are doing pretty good. But how did I arrive (0,2,4,6)? The answer is by using a *sampling ratio* which is nothing more than an interpolation factor. Here's the math:

In general,

$$\text{sampling\_ratio} = \text{source\_height} / \text{destination\_height}$$

Thus the sampling ratio is,

```
sampling_ratio = 8/4 = 2.
```

Thus, every 1 pixel we move on the destination polygon in the vertical axis, we must move 2 pixels on the source to keep up. That's where the 2 comes from and hence the sampling sequence (0,2,4,6). If you're still with me then you should say wait a minute, we lost information! And indeed we did, we had to throw away half the pixels. This is definitely a problem with sampling on an integer matrix without any averaging. If you were writing a high end 3D modeler like 3D Studio MAX then you would probably average the pixels your sampling (area sampling), so as to get a better approximation, but for games and real-time our technique will do. Now let's see another example of the opposite case.

## Example 2

In example 1, we saw that the source texture was compressed, that is, the destination was smaller than the source, thus information was lost. The second case of course would be when the destination is bigger than the source and there isn't enough information to go around. In this case, the source data must be sampled more than once and replicated. This is where all "chunkyness" comes from when texture mapped polygons get too close to you in a 3D game. There isn't enough texture data so some sample points are sampled many times creating big blocks. Anyway, referring to the second example in Figure 5.0, we see that the source is again 1x8, but this time the destination is 1x14 pixels, yuck! Obviously, we are going to need a fractional sampling ratio, but let's let the math do it:

As usual,

```
sampling_ratio = source_height / destination_height;
```

Thus the sampling ratio is,

```
sampling_ratio = 8/14 = 0.57
```

Hence, every pixel we draw on the destination polygon we should sample it 0.57 units from the last sample point on the source. This gives us the following sample point sequence for destination pixels (0,1,2,3,...,13):

```
Sample 0: 0.57
Sample 1: 1.14
Sample 2: 1.71
Sample 3: 2.28
Sample 4: 2.85
Sample 5: 3.42
Sample 6: 3.99
Sample 7: 4.56
Sample 8: 5.13
Sample 9: 5.7
Sample 10: 6.27
Sample 11: 6.84
Sample 12: 7.41
Sample 13: 7.98
```

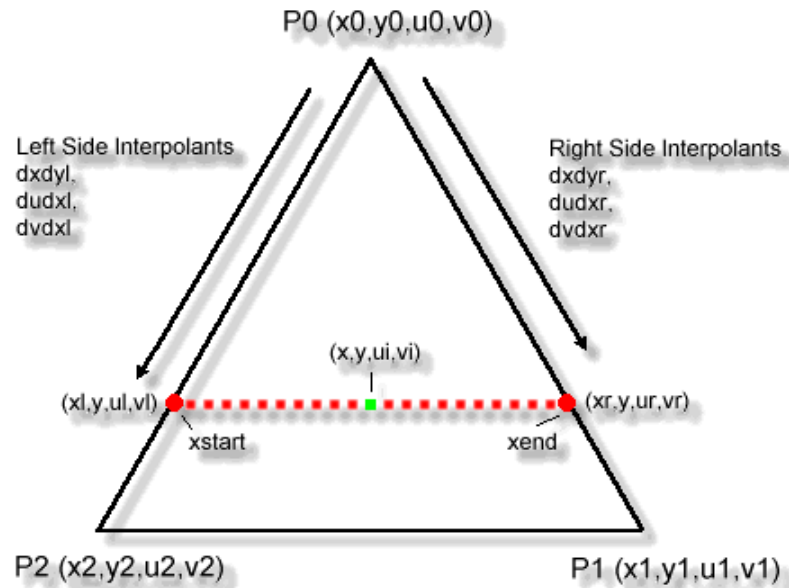
And to get the actual sample points, we simply truncate the sample points in integer space or take the *floor* of each value resulting in the sample points (0,1,1,2,2,3,3,4,5,5,6,6,7,7) which sounds about right. Each point got sampled about 2 times, or  $1/0.57 \sim 2.0$ . That's what I'm talking about!

## Multiple Interpolations

When I wrote my first affine texture mapper I thought that something must be wrong since it seemed like I was interpolating everything, but the kitchen sink! However, the truth is, there is really no way around all the various interpolants, and in the end the inner loop for each pixel can be optimized into around 10 cycles per pixel on a Pentium which translates to a theoretical max of 10 - 20 million textels (textured pixels) per

second on a Pentium 100mhz which in reality will be far less than that due to a million reasons such as the caches, memory bandwidth, video card, etc. Now, let's talk about the algorithm in general and then derive the math for it.

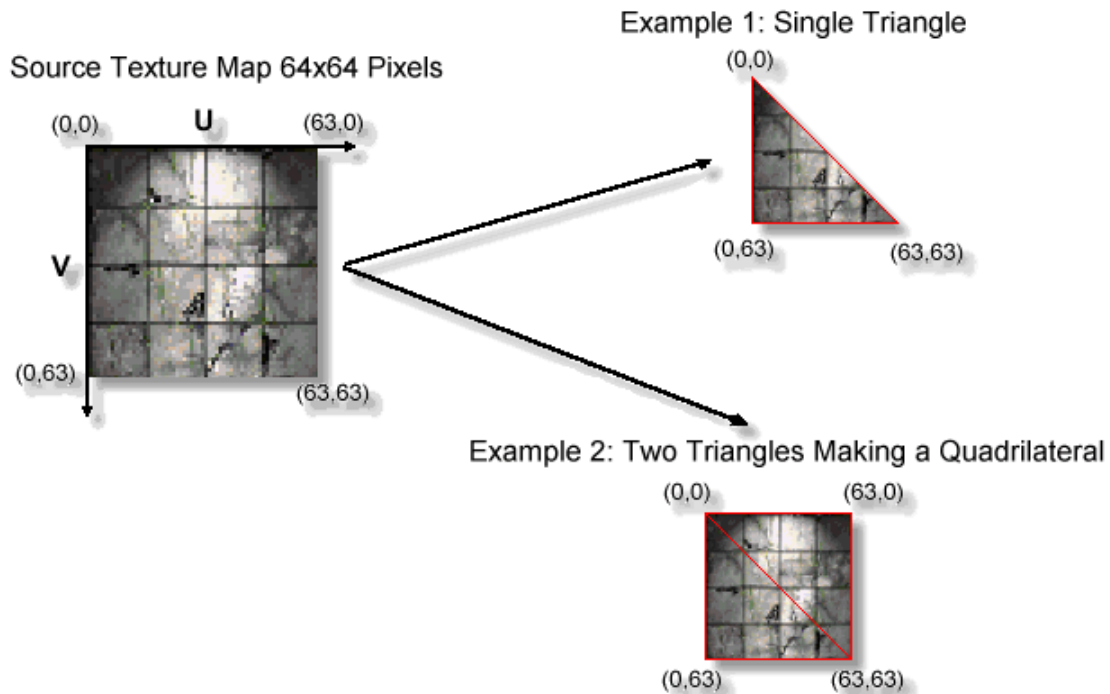
Figure 6.0 - Graphical Representation of Texture Mapping Algorithm.



The idea behind the algorithm is that we want to interpolate down the left and right edges of the triangle and draw each scanline strip as we go with the proper texture pixels. So what we need to do first is assign full texture coordinates to the vertices of the destination triangle to give us a frame of reference for the interpolants. Thus we must assign each vertex a  $(u,v)$  texture coordinate as shown in Figure 6.0. Therefore, each vertex has a total of 4 data components or it's a 4-D value -- weird huh? Moving on, let's talk about the range of the texture coordinates. Since our source texture map is 64x64 pixels that means that the texture coordinates must range from 0 - 63 for any vertex. This will map or stretch the texture map to each vertex.



Figure 7.0 - Texture Mapping 3 and 4 Sided Polygons.



For example, in Figure 7.0 we see a couple examples: one triangle with the texture coordinates (0,0), (63,0), and (63,63) mapped to vertices 0,1, and 2 respectively. This basically copies half of the texture map to the destination triangle which is what we would expect. In the second example in Figure 7.0 we see the same texture mapped onto 2 triangles which are adjacent to each other forming a square. In this case, the texture coordinates are selected in such a way that half of the texture map is mapped to one triangle and the rest to the other, hence, a perfect texture wrapping around two triangles. Moreover, this is how you would make a quadrilateral, that is, with two triangles. Now that you have a visual on the problem and know the labeling from Figure 6.0, let's implement the algorithm mathematically. *Note that the variable names used in the following analysis are based on Figure 6.0 and the final program, so that you can follow the program code more easily.*

The left edge interpolants are:

```

dx dy1 = (x2 - x0) / (y2 - y0); // the x interpolant for the left hand side
du dy1 = (u2 - u0) / (y2 - y0); // the u interpolant for the left hand side
dv dy1 = (v2 - v0) / (y2 - y0); // the v interpolant for the left hand side

```

and similarly the right edge interpolants are,

```

dx dyr = (x1 - x0) / (y2 - y0); // the x interpolant for the right hand side
du dyr = (u1 - u0) / (y2 - y0); // the u interpolant for the right hand side
dv dyr = (v1 - v0) / (y2 - y0); // the v interpolant for the right hand side

```

Of course, there's a lot of room for optimization in the math, for example, the term (y2 - y0) is common and need only be computed once, furthermore, it's better to compute the reciprocal of (y2 - y0) and then multiply, but you get the idea. Anyway, now that we have the interpolants we are almost ready to rock. The interpolants must be in reference to some starting point, right? This starting is the topmost vertex, vertex 0. Hence, we need to start the algorithm off like this:

```

xl = x0; // the starting point for the left hand side edge x interpolation
ul = u0; // the starting point for the left hand side edge u interpolation
vl = v0; // the starting point for the left hand side edge v interpolation

```

And for the right hand side,

```

xr = x0; // the starting point for the right hand side edge x interpolation
ur = u0; // the starting point for the right hand side edge u interpolation
vr = v0; // the starting point for the right hand side edge v interpolation

```

Now we are almost ready to go, we can interpolate down the left hand edge and the right edge with:

```

xl+=dxdyl;
ul+=dudyl;
vl+=dvdyl;

```

and,

```

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;

```

But at each point on the left and right hand edge of the triangle we still need to perform once more linear interpolation across the scanline! This is the final interpolation and the one that will give us our texture coordinates (ui,vi) which we will use as [row, column] indices into the texture bitmap to obtain the texel. All we need to do is compute the u,v coordinate on the left and right side and then use the dx to compute a linear interpolation factor for each. Here's the math::

```

dx = (xend - xstart); // the difference or delta dx
xstart = xl;           // left hand starting point
xend = xr;             // right hand starting point

```

Therefore, the interpolants across each scanline in u,v space are,

```

du = (ul - ur)/dx;
dv = (vl - vr)/dx;

```

Then with du,dv, we have everything we need to interpolate across the scanline at vertical position y from xstart to xend. Here's a code fragment:

```

// initialize u,v interpolants to left and right hand side values
ui = ul;
vi = vl;

// now interpolate from left to right, i.e, in a positive x direction
for (x = xstart; x <= xend; x++)
{
    // get texture pixel value
    pixel = texture_map[ui][vi];

    // plot pixel at x,y
    Plot_Pixel(x,y,pixel);

    // advance u,v interpolants
    ui+=du;
    vi+=dv;
} // end for x

```

That's it. Of course for the outer loop you would still interpolate xl,ul,vl,xr,ur,vr down the triangle edges for each scanline of the triangle.

The final code for the texture mapper is shown in Listing 2.0 and 3.0, the function assumes a specific input data structure and that the texture map is a linear bitmap 64x64 pixels, but other than that, it's nothing more

than an implementation of our derivation here along with all the triangle cases, and clipping, so it doesn't blow up. In addition, Listing 4.0 is a complete DirectX demo of the texture mapper in action. It draws random triangles all over the screen in 640x480x256.

**Listing 2.0 - Header file for Affine texture mapper. ([production insert tmapper.h](#))**

**Listing 3.0 - Affine texture mapper. ([production - insert tmapper.cpp](#))**

**Listing 4.0 - A complete DirectX demo of the texture mapper. ([production - insert tmapdemo.cpp](#))**

If you don't want to type the code, you can find it along with the executable within **TMAPSRC.ZIP** which you can download from ([production insert FTP url here](#)). In addition, the demo uses fixed point math even though we used floating point here. I have found that all math should be done using floating point, until final rasterization (as in this case) since the conversion from floating point to integer and visa-versa kills you. Finally, as a test of the texture mapper, I gave it to a friend of mine -- **Jarrold Davis** and he created a 3D demo by adding the texture mapper to his flat shaded 3D engine, thanks Jarrod. The demo is called **BOX2.EXE** and is within the **TMAPSRC.ZIP** archive file as well, so enjoy.

Well, that's it for texture mapping. Maybe next time we can talk about lighting those texture mapped polygons...