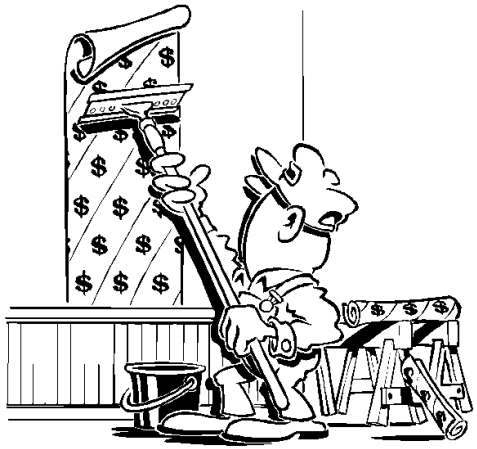
Texture Mapping

- Why texture map?
- How to do it
- How to do it right
- Spilling the beans
- A couple tricks
- News on Project #4



The Quest for Visual Realism

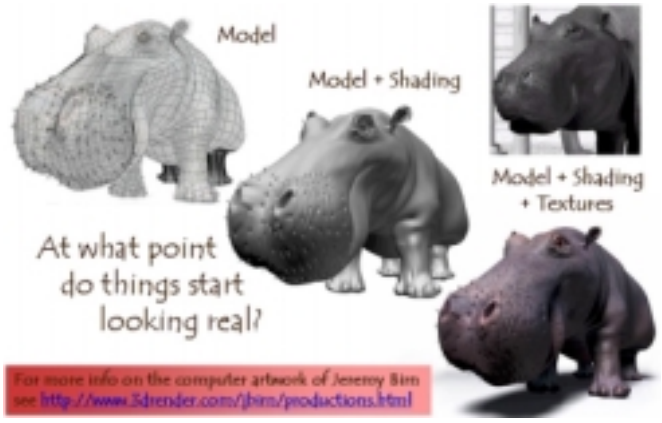
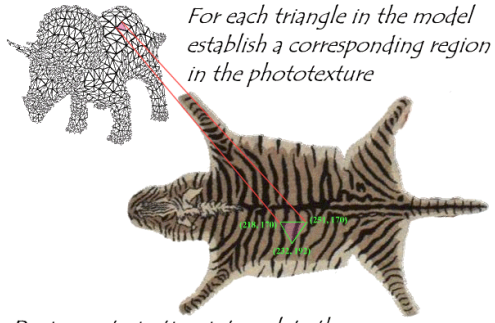


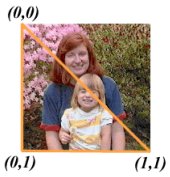
Photo-textures

The concept is very simple!



Texture Coordinates

- Specify a texture coordinate at each vertex *(s, t)* or *(u, v)*
- Canonical coordinates where *u* and *v* are between 0 and 1
- Simple modifications to triangle rasterizer




```
public void Draw(Raster raster) {
    :
    PlaneEqn(uPlane, u0, u1, u2);
    PlaneEqn(vPlane, v0, v1, v2);
    :
    for (y = yMin; y <= yMax; y += raster.width) {
        e0 = t0;  e1 = t1;  e2 = t2;
        u = tu;   v = tv;   z = tz;
        boolean beenInside = false;
        for (x = xMin; x <= xMax; x++) {
            if ((e0 >= 0) && (e1 >= 0) && (e2 >= 0)) {
                int iz = (int) z;
                if (iz <= raster.zbuff[y+x]) {
                    int uval = u * texture.width;
                    uval = tile(uval, texture.width);
                    int vval = v  * texture.height;
                    vval = tile(vval, texture.height);
                    int pix = texture.getPixel(uval, vval);
                    if ((pix & 0xff000000) != 0) {
                        raster.pixel[y+x] = pix;
                        raster.zbuff[y+x] = iz;
                    }
                }
                beenInside = true;
            } else if (beenInside) break;
            e0 += A0;   e1 += A1;   e2 += A2;
            z += Az;    u += Au;    v += Av;
        }
        t0 += B0;   t1 += B1;   t2 += B2;
        tz += Bz;   tu += Bu;   tv += Bv;
    }
}
```

The Result

Click and drag on the applet above to rotate the texture mapped cube.

Wait a minute... that doesn't look right.
What's going on here?

Another Example

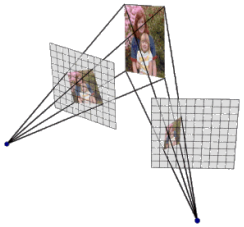
Let's try again with a simpler texture...

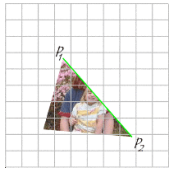
Click and drag on the applet above to rotate the texture mapped cube.

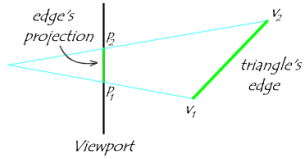
Notice how the texture seems to bend and warp along the diagonal triangle edges. Let's take a closer look at what is going on.

Looking at One Edge

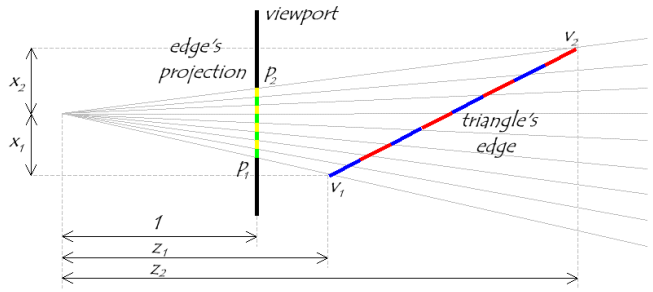
First, let's consider one edge from a given triangle. This edge and its projection onto our viewport lie in a single common plane. For the moment, let's look only at that plane, which is illustrated below:







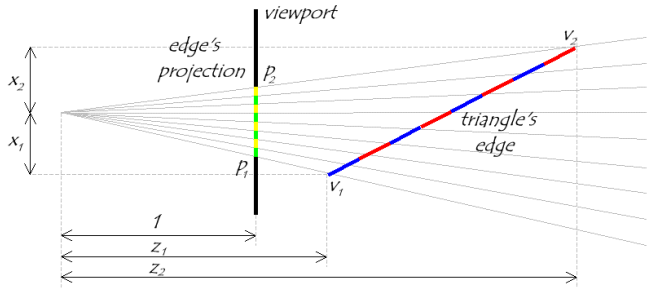
Visualizing the Problem



Notice that uniform steps on the image plane do not correspond to uniform steps along the edge.

WLOG, let's assume that the viewport is located 1 unit away from the center of projection.

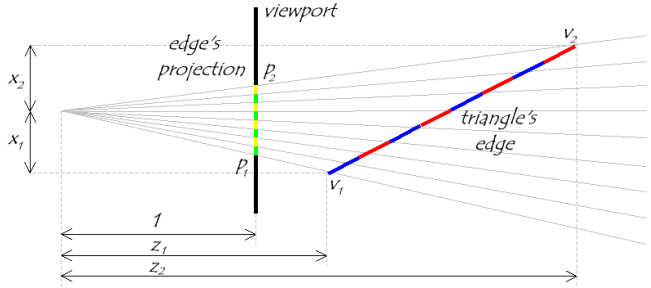
Linear Interpolation in Screen Space



Compare linear interpolation in screen space

$$p(t) = p_1 + t(p_2 - p_1) = \frac{x_1}{z_1} + t\left(\frac{x_2}{z_2} - \frac{x_1}{z_1}\right)$$

Linear Interpolation in 3-Space



to interpolation in 3-space

$$\begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} x_1 \\ z_1 \end{bmatrix} + s\left(\begin{bmatrix} x_2 \\ z_2 \end{bmatrix} - \begin{bmatrix} x_1 \\ z_1 \end{bmatrix}\right) \qquad P\left(\begin{bmatrix} x \\ z \end{bmatrix}\right) = \frac{x_1 + s(x_2 - x_1)}{z_1 + s(z_2 - z_1)}$$

How to Make Them Mesh

Still need to scan convert in screen space... so we need a mapping from *t* values to *s* values.

We know that the all points on the 3-space edge project onto our screen-space line. Thus we can set up the following equality:

$$\frac{x_1}{z_1} + t\left(\frac{x_2}{z_2} - \frac{x_1}{z_1}\right) = \frac{x_1 + s(x_2 - x_1)}{z_1 + s(z_2 - z_1)}$$

and solve for *s* in terms of *t* giving:

$$s = \frac{t\, z_1}{z_2 + t\,(z_1 - z_2)}$$

Unfortunately, at this point in the pipeline (after projection) we no longer have z_1 and z_2 lingering around (Why?). However, we do have $w_1 = 1/z_1$ and $w_2 = 1/z_2$.

$$s = \frac{t\,\frac{1}{w_1}}{\frac{1}{w_2} + t\,(\frac{1}{w_1} - \frac{1}{w_2})} = \frac{t\, w_2}{w_1 + t\,(w_2 - w_1)}$$

Interpolating Parameters

We can now use this expression for *s* to interpolate arbitrary parameters, such as texture indices (*u*, *v*), over our 3-space triangle. This is accomplished by substituting our solution for *s* given *t* into the parameter interpolation.

$$u = u_1 + s(u_2 - u_1)$$

$$u = u_1 + \frac{t\, w_2}{w_1 + t\,(w_2 - w_1)}(u_2 - u_1) = \frac{u_1 w_1 + t\,(u_2 w_2 - u_1 w_1)}{w_1 + t\,(w_2 - w_1)}$$

Therefore, if we **premultiply all parameters that we wish to interpolate in 3-space by their corresponding *w* value** and add a new plane equation to interpolate the *w* values themselves, we can interpolate the numerators and denominator in screen-space. We then need to perform a divide a each step to get to map the screen-space interpolants to their corresponding 3-space values.

Once more, this is a simple modification to our existing triangle rasterizer.

Modified Triangle Code

```
    .
    .
PlaneEqn(uPlane, (u0*w0), (u1*w1), (u2*w2));
PlaneEqn(vPlane, (v0*w0), (v1*w1), (v2*w2));
PlaneEqn(wPlane, w0, w1, w2);
    .
    .
for (y = yMin; y <= yMax; y += raster.width) {
    e0 = t0;        e1 = t1;        e2 = t2;
    u = tu;         v = tv;         w = tw;        z = tz;
    boolean beenInside = false;
    for (x = xMin; x <= xMax; x++) {
        if ((e0 >= 0) && (e1 >= 0) && (e2 >= 0))) {
            int iz = (int) z;
            if (iz <= raster.zbuff[y+x]) {
                float denom = 1.0f / w;
                int uval = (int) (u * denom + 0.5f);
                uval = tile(uval, texture.width);
                int vval = (int) (v * denom + 0.5f);
                vval = tile(vval, texture.height);
                int pix = texture.getPixel(uval, vval);
                if ((pix & 0xff000000) != 0) {
                    raster.pixel[y+x] = pix;
                    raster.zbuff[y+x] = iz;
                }
             }
             beenInside = true;
         } else if (beenInside) break;
         e0 += A0;    e1 += A1;    e2 += A2;
         z += Az;     u += Au;     v += Av;     w += Aw;
    }
    t0 += B0;    t1 += B1;    t2 += B2;
    tz += Bz;    tu += Bu;    tv += Bv;    tw += Bw;
}
```


Demonstration

For obvious reasons this method of interpolation is called *perspective-correct interpolation*. The fact is, the name could be shortened to simply *correct interpolation*. You should be aware that not all 3-D graphics APIs implement perspective-correct interpolation.

Clicking and dragging on these cubes should give a more reasonable result.

Dealing with Incorrect Interpolation

You can reduce the perceived artifacts of non-perspective correct interpolation by subdividing the texture-mapped triangles into smaller triangles (why does this work?). But, fundamentally the screen-space interpolation of projected parameters is inherently flawed.

In this example the front face of the right cube is subdivided into 8 triangles and screen-space interpolation is used. While the artifacts of this approximation are less obvious, they can still be seen as you move around the cubes.

Wait a Minute!

When we did Gouraud shading didn't we interpolate illumination values that we found at each vertex using *screen-space* interpolation?

Didn't I just say that screen-space interpolation is wrong
(I believe "inherently flawed" were my exact words)?

Does that mean that Gouraud shading is wrong?

Is everything that I've been telling you all one big lie?

Has 6.837 amounted to a total waste of time?

Yes, Yes, Yes, Maybe, and

No, you've been exposed to nice purple cows.

Gourand shading is wrong. However, you usually will not notice because the transition in colors is very smooth (And we don't know what the right color should be anyway, all we care about is a pretty picture).

There are some cases where the errors in Gourand shading become obvious:

- When switching between different levels-of-detail representations
- At "T" joints.

A "T" joint

Texture Tiling

Often it is useful to *repeat* or *tile* a texture over the surface of a polygon. This was implemented in the tile method of the examples that I gave.

```
                .
                .
                .
        float denom = 1.0f / w;
        int uval = (int) (u * denom + 0.5f);
        uval = tile(uval, texture.width);
        int vval = (int) (v * denom + 0.5f);
        vval = tile(vval, texture.height);
                .
                .
                .

private int tile(int val, int size) {
    if (val >= size) {
        do { val -= size; } while (val >= size);
    } else {
        while (val < 0) { val += size; }
    }
}
```


Texture Transparency

There was also a little code snippet to handle texture transparency.

```
                .
                .
                .
int pix = texture.getPixel(uval, vval);
if ((pix & 0xff000000) != 0) {
    raster.pixel[y+x] = pix;
    raster.zbuff[y+x] = iz;
}
                .
                .
                .
```

Now you can all go out and write DOOM!

Here is Where I Give Away Project #4

1. All vertices sent to the Triangle's ClipAndDraw( ) method are *unnormalized* and in a *canonical clipping space*. You might want to take advantage of this when clipping.
2. *hither* = *near*, *yon* = *far* (Eventually I will find all of these throughout my notes)
3. You must implement the transformation of vertex normals, although you won't need them unless you choose to implement per-vertex shading.
4. The box sits above a plane. It looks weird because clipping is not implemented. Use this example to test your clipping and culling code. To test culling, I suggest that you comment out a cube face (place a # in front of the line).
5. Due date is extended to midnight of next Wednesday (11/22).
6. Any other questions?

Next Time

Ray Tracing...
but for now, let's play Doom!