TEXTURE MAPPING POLYGONS IN PERSPECTIVE

Paul Heckbert

Computer Graphics Lab
New York Institute of Technology

Technical Memo No. 13
28 April 1983

1. INTRODUCTION

Mapping textures onto polygons is more difficult for perspective projections than for parallel ones. Texture mapping in perspective requires a division at each pixel in the general case, while parallel projections require none. Simple incremental formulas presented here make perspective texture mapping efficient for the inner loop of a polygon renderer.

We will also discuss the use of a "mipmap" for efficient, approximate antialiasing of textures. Use of a mipmap requires the calculation of a variable "d" which is a measure of the area in the texture to be filtered. We will discuss several formulas for "d" which apply to many surface types (not just polygons). We show how one of these formulas can be efficiently computed for polygons; it is currently in use by my POLY program.

Notation:

(x,y) is a point in screen space (frame buffer coords)

(u,v) is a point in texture space

d is proportional to the diameter of the area in the texture to be filtered

points are represented by 3 or 4 element column vectors

points are transformed by matrix multiplication with a 4x3 or 4x4 transformation matrix on the left

2. TEXTURE MAPPING POLYGONS IN PERSPECTIVE

Mapping Texture Space to Object Space

When mapping a texture onto a polygon in 3D, you usually specify the mapping by setting up a correspondence between points in object space (the space in which the polygon vertices are defined) and points in texture space.

In my program POLY, this is done by specifying the u and v coordinates corresponding to each vertex of the polygon. For simplicity of implementation, the mapping between texture space and object space is assumed to be affine (to consist of only scales, rotations, and translations). You can map a rectangle

to a parallelogram (or vice versa), but you can't map a rectangle to a trapezoid. Since a 3D affine transformation has six degrees of freedom, three points can determine it. This affine texture space to object space transformation can be written as follows:

Mapping Object Space to Screen Space

The standard formula for mapping object space to screen space is best expressed with a 4x4 matrix using homogeneous coordinates:

Mapping Texture Space to Screen Space

A perspective mapping of texture space to screen space can be computed by concatenating the two matrices above:

Z is irrelevant, since it is not needed for texture mapping. Note that we can choose I=1 with no loss of generality.

This transformation maps a rectangle in texture space into a quadrilateral in screen space.

For synthesized animation, you usually know the object to screen space transform. In interactive or motion-tracking applications, however, you might want to define the texture space to screen space mapping by giving the screen-space coordinates of the vertices of a polygon. Four points define a perspective texture to screen space transform, since they have eight degrees of freedom, and the equations above have eight variables.

The transform can be found very simply by setting up the following equations for each of the four points:

This forms an 8x8 system of equations in the variables A-G [2]. This is the method used by my PWARP program.

For parallel projections, the eye is infinitely far away, so w' is constant, and the formulas reduce to an affine transformation:

Mapping Screen Space to Texture Space

When texture mapping a polygon, you usually scan it out in screen space and compute u and v from x and y. This means we'll need the inverse of the mappings above. The inverse mapping has the same form as the original:

A rectangle in screen space is mapped to a quadrilateral in texture space. For parallel projections, the quadrilateral is a parallelogram.

For a given scan line, y is constant, so most of the terms are constant. The two numerators and one denominator can be evaluated once at the left end of the scan segment and computed incrementally in the x loop. This requires only three additions and two divisions per pixel.

Inner loop of a polygon texture-mapper which samples:

```
x = xleft
unum = t0*x+t1*y+t2
vnum = t3*x+t4*y+t5
den  = t6*x+t7*y+t8
for (; x<=xright; x++) {
   sample the point (unum/den,vnum/den) in texture
   write texture color to (x,y) in screen
   unum += t0
   vnum += t3
   den  += t6
}
```

3. TEXTURE MAPPING WITH MIP

What is a Mipmap?

Below is a brief description of mipmaps.

At NYIT, textures are usually in the form of a "mipmap", which is a frame buffer containing the red, green, and blue components of a texture or image in the upper right, lower right, and lower left

quadrants, respectively. Each component is filtered down to 9 different scales, and the 27 pictures fit nicely into a 512x512 frame buffer. Mipmaps are currently made by repeatedly halving the size of the picture, starting with the 512x512 original, averaging 2x2 blocks of pixels on one level to compute the pixel values on the next level up. This averaging has the same effect as convolution with a square box filter.

If you stack the maps they form a pyramid. We'll use the variable "d" to measure height in this pyramid. Let's label the original 512x512 image with d=1 (even though it is not in most mipmaps), the 256x256 image with d=2, ..., up to the tiny 1x1 image, which will have d=512.

To sample a texture using a mipmap, the variable d selects which of the nine sets of maps is referenced, and (u,v) selects the pixel to be pulled from that map. The current implementation actually reads 8 pixels per component (four from each of two maps) and performs trilinear interpolation among them. So for the constant cost of 24 pixel reads and 21 (or more) multiplications, you can approximately filter any square window of the texture.

The standard alternative to "mipmapping" is straightforward filtering of the texture every time you need a sample. This can be quite slow, especially when scaling a texture way down.

How to choose d

We'll assume that most surfaces (whether bicubic, quadric, or whatever) are approximately planar over the area covered by a pixel. Under this approximation, a rectangular screen space pixel is mapped into a quadrilateral in texture space. We want to filter the quadrilateral to compute the average texture color for the current pixel. For most pixels you can further assume the quadrilateral is nearly a parallelogram. This is a valid assumption for pixels which are not near a vanishing point or a singularity of the texture. (If all pixels map to parallelograms, the surface is planar and parallel-projected).

The size of the approximating parallelogram is given by four partial derivatives:

The vertices of the quadrilateral can be found exactly by computing differences between the u's and v's of adjacent pixels.

The standard mipmap access cannot filter an arbitrary quadrilateral (or even an arbitrary rectangle). It can only "properly" filter square areas. We thus have the problem of choosing a "good" value for d.

With the numbering system described above, d is the side length ("diameter") of the squares in the original texture to be averaged. We need to approximate a quadrilateral with a square in order to choose a value for d. If our square is too small, the filter will be too sharp, and the mapped texture jaggy. If the square is too large, the mapped texture will be unnecessarily blurry.

Choosing a formula for d is a problem in heuristics and approximation. Here are some of the formulas for d which I have dreamed up:

find box around
quadrilateral:

box approximating
parallelogram:

find lengths of x and y
direction vectors:

4. COMPUTING D FOR POLYGON TEXTURE MAPPING

For a polygon in perspective, the four partial derivatives simplify to:

Note that the numerators of the x terms are functions of y only, and the numerators of the y terms functions of x only. This simplifies many of the formulas for d substantially.

We discard the area-related formulas, since they will cause aliasing when the quadrilateral is thin (we'd rather be blurry than jaggy). To be on the safe side, we can further dismiss the formulas which use an average rather than a maximum, since they will underfilter along the longer dimension of a rectangular area.

From the remaining candidates, we pick formula (7). It can be calculated as follows:

If we pre-compute ylen for the polygon's range of x's and save these in an array, and compute xlen once per scan line, we can avoid the sqrt at each pixel! The cost of this formula is one max function, one multiply, and one divide at each pixel, which is very reasonable (much faster than the mipmap access, in fact).

This is the formula currently in use by POLY. As you can see in figure 3, it blurs too much near the horizon. The blurriness cannot be helped much by choosing a different formula. When texture mapping a polygon like the one in this figure, we need to filter long, thin strips in the texture. We have run into the fundamental limitation of mipmaps: they are intended for square filters. Experiments have shown that less conservative formulas (eg. (6)) cause aliasing. Based on these experiments, I have concluded that (7) is the best formula for d.

5. REFERENCES

[1] Smith, Alvy Ray, Incremental Rendering of Textures in Perspective, SIGGRAPH '80 Animation Graphics Tutorial Notes.

[2] Thornton, Robert, Model: Interactive Modeling in Three Dimensions Through Two-Dimensional Windows, MS Thesis, Cornell U., 1977, p. 17

[3] /usr1/e/movi/trin.c

[4] Williams, Lance, Pyramidal Parametrics, SIGGRAPH '83 Proceedings

[5] Newman, William M., and Robert F. Sproull, Principles of Interactive Computer Graphics, 2nd. ed. McGraw Hill, NY. 1979

[6] Carlblom, Ingrid, and Joseph Paciorek, Planar Geometric Projections and Viewing Transformations, Computing Surveys, Vol. 10, No. 4, Dec. 1978

[7] /usr/grp/3d/poly/texinit.c