Interpolation for Polygon Texture Mapping and Shading*

Paul S. Heckbert
Henry P. Moreton†

Dept. of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720
USA

Abstract

A simple, fast method is presented for the interpolation of texture coordinates and shading parameters for polygons viewed in perspective. The method has application in scan conversion algorithms like z-buffer and painter's algorithms that perform screen space interpolation of shading parameters such as texture coordinates, colors, and normal vectors. Some previous methods perform linear interpolation in screen space, but this is rotationally variant, and in the case of texture mapping, causes a disturbing "rubber sheet" effect. To correctly compute the nonlinear, projective transformation between screen space and parameter space, we use *rational linear interpolation* across the polygon, performing several divisions at each pixel. We present simpler formulas for setting up these interpolation computations, reducing the setup cost per polygon to nil and reducing the cost per vertex to a handful of divisions.

Additional keywords: incremental, perspective, projective, affine.

1 Introduction

We first define our terminology, then summarize a naive, linear method for interpolating shading parameters during scan conversion. After examining the flaws of linear interpolation, we describe the new method and prove its correctness. Readers uninterested in the proofs may want to read just the sections titled "Polygon Rendering with Linear Interpolation" and "New Algorithm".

*Appeared in *State of the Art in Computer Graphics: Visualization and Modeling*, David F. Rogers and Rae A. Earnshaw, eds., Springer-Verlag, New York, 1991, pp. 101-111. (Proc. of State of the Art in Computer Graphics Summer Institute, Edinburgh, July 1990).

†Second author supported by a fellowship from Silicon Graphics. email: ph@miro.berkeley.edu, moreton@fezzik.berkeley.edu

1.1 Definitions

We define the following coordinate systems: *Object space* is the 3-D coordinate system in which each polygon is defined. There can be several object spaces. *World space* is a coordinate system that is related to each object space by 3-D modeling transformations (translations, rotations, and scales). *3-D screen space* is the 3-D coordinate system of the display, a "perspective space" with pixel coordinates (x, y) and depth z. It is related to world space by the camera parameters. Finally, *2-D screen space* (or "screen space" for short) is the 2-D subspace of 3-D screen space without z.

To facilitate affine and projective (perspective) transformations, we use homogeneous notation [Maxwell46] in which, for example, the 2-D real point (x, y) is represented by the 3-D homogeneous vector $\mathbf{p} = (xw, yw, w)$, where w is an arbitrary nonzero number. We will be cavalier about treating the case where $w = 0$. In homogeneous notation, 2-D points are represented by 3-vectors and 3-D points are represented by 4-vectors.

We use the following notation:

COORDINATE SYSTEM	REAL	HOMOGENEOUS
3-D object space	(x_o, y_o, z_o)	$\mathbf{p_o} = (x_o w_o, y_o w_o, z_o w_o, w_o)$
3-D screen space	(x, y, z)	$\mathbf{p}_\sigma = (xw, yw, zw, w)$
2-D screen space	(x, y)	$\mathbf{p_s} = (xw, yw, w)$

1.2 Projective and Affine Mappings

We will use two classes of mapping (transformation): affine and projective. The 3-D forms of these mappings are ubiquitous in computer graphics [Newman-Sproull76]. The 2-D *projective mapping* (or perspective mapping) from (u, v) to (x, y) has the general form [Maxwell46]:

$$x = \frac{au + bv + c}{gu + hv + i}, \quad y = \frac{du + ev + f}{gu + hv + i}$$

The mapping is more simply represented in homogeneous matrix notation:

$$\begin{pmatrix} xw & yw & w \end{pmatrix} = \begin{pmatrix} uq & vq & q \end{pmatrix} \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$$

Affine mappings include scales, rotations, translations, and shears; they are linear mappings plus a translation. A 2-D projective mapping is affine iff $g = h = 0$ and $i \neq 0$.

These mappings are trivially generalized to map an m-dimensional space to an n-dimensional space. The homogeneous matrix for such a mapping would be $(m+1) \times (n+1)$. Unlike affine mappings, projective mappings do not preserve parallel lines or equispaced points along a line, but like affine mappings, projective mappings preserve lines, that is, lines transform to lines. Projective mappings are closed under composition: they may be composed by concatenating their matrices. Projective mappings between spaces of equal dimension are invertible using the inverse or adjoint matrix.

We call a parameter or space *X-affine* when it is an affine function or transform of space X, and *X-projective* when it is an projective function or transform of space X. For example, in texture mapping, texture space is typically object-affine, but screen-projective.

2 Polygon Rendering with Linear Interpolation

Scan conversion algorithms such as z-buffer, painter's, and scanline methods [Rogers85] typically use a set of interpolated shading parameters at each pixel. This set might include: texture coordinates (u, v) for texture mapping [Blinn-Newell76], [Heckbert86], (r, g, b) for Gouraud shading, a normal vector for Phong shading, and world space position for per-pixel shading. Polygons are described by listing these parameter values along with the object space coordinates (x_o, y_o, z_o) at each vertex. During scan conversion, both the parameters and the screen coordinates (x, y, z) are interpolated along the edges of the polygon from scanline to scanline, and then interpolated across each scanline for use at each pixel.

The steps of the linear interpolation algorithm are:

(1) Associate a record containing the parameters of interest with each vertex of the polygon.

(2) For each vertex, transform object space coordinates to homogeneous screen space using 4×4 object to screen matrix, yielding the values (xw, yw, zw, w).

(3) Clip the polygon against plane equations for each of the six sides of the viewing frustum, linearly interpolating all the parameters when new vertices are created.

(4) Perform a homogeneous division to compute $x = xw/w$, $y = yw/w$, $z = zw/w$.

(5) Scan convert in screen space, by linear interpolation of all parameters, using the parameter values at each pixel for shading.

C code for such a generic polygon clipper and scan converter is available in [Heckbert90].

2.1 Flaws of Linear Interpolation

Linear interpolation algorithms like the above are usually used for Gouraud shading, Phong shading, and often for texture mapping as well. It is wrong, however, to perform linear interpolation in screen space of parameters that are not screen-affine. We assume that the only perspective in the transformation pipeline lies between world space and screen space. That is: world space, object space, and the parameters are mutually affine, but they are screen-projective. The above algorithm is correct only when the parameters are screen-affine, which occurs only for parallel projection or for perspective projection of a plane perpendicular to the line of sight.

The flaws are most visible in texture mapping. Figure 2 shows the artifacts of linear, screen space interpolation of texture coordinates. Note that this image does not exhibit the foreshortening we expect from perspective. The texture also shows disturbing discontinuities along horizontal lines passing through the vertices. In animation, the horizontal ripples will

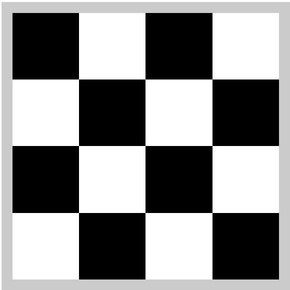

Figure 1: *A checkerboard texture.*

move distractingly as the camera rolls, since the ripples are rotation variant, and the lack of foreshortening will make the texture appear to slide across the surface like a rubber sheet. Figure 3 shows the correct image.

The above problems occur because the texture transformation effected by our linear parameter interpolation is inconsistent with the geometry transformation used to transform the vertices to screen space. Linear interpolation computes a piecewise bilinear mapping from screen space to parameter space, while the actual mapping defined by affine transformations and a perspective camera is projective. (A bilinear mapping from (x, y) to (u, v) is one of the form $u = axy + bx + cy + d$, $v = exy + fx + gy + h$).

Similar errors occur when colors, normals, or positions are linearly interpolated in a space to which they are not affine, but these errors are much less noticeable than the errors for texture mapping. The flaws in Gouraud and Phong shading are so subtle, in fact, that they went unnoticed for several years in the production renderer at the New York Institute of Technology (actually, it is hard to say what "correct" interpolation means for Gouraud and Phong shading, since they are approximations).

The rubber sheet effect occurs for polygons with any number of sides, but rotational variation occurs only for polygons with four or more sides. Linear interpolation across polygons in screen space effects an affine mapping, which is rotation invariant. Rotation invariance does not imply correctness, however: if the scene employs affine texture-to-object parameterization and a perspective camera, then texture space is screen-projective, not screen-affine, and linear interpolation is incorrect. Although triangulation is usually inappropriate, it is often used to solve the rotational variation problem.

The correct solution described later involves several divisions per pixel. A cheap alternative that avoids divisions at each pixel is polygon subdivision, where parameter values at the new vertices are computed using linear interpolation in object space. The Silicon Graphics VGX machine currently does texture mapping this way. In figure 4 we can see how splitting a polygon into a number of smaller polygons improves the approximation. Others have approximated the nonlinear function with quadratic or cubic polynomials [Wolberg90]. But note that rational linear functions have poles (behaving like $f(x) = 1/x$) at the horizon of an infinite plane, near which they are not well approximated by linear functions or other polynomials. If subdivision is used, it should be adaptive. We do not recommend subdivision or approximation, however, as they increase the number of polygons, and the results

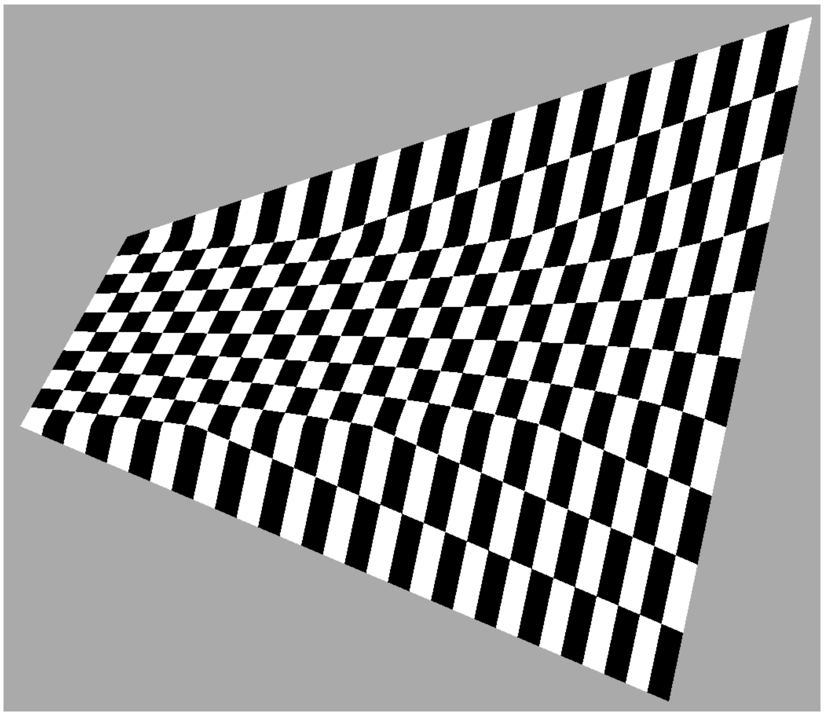

Figure 2: *Image produced by texture mapping the checkerboard onto a rectangle in perspective using linear interpolation of* u, v. *Note horizontal lines of discontinuity passing through the vertices on the left.*

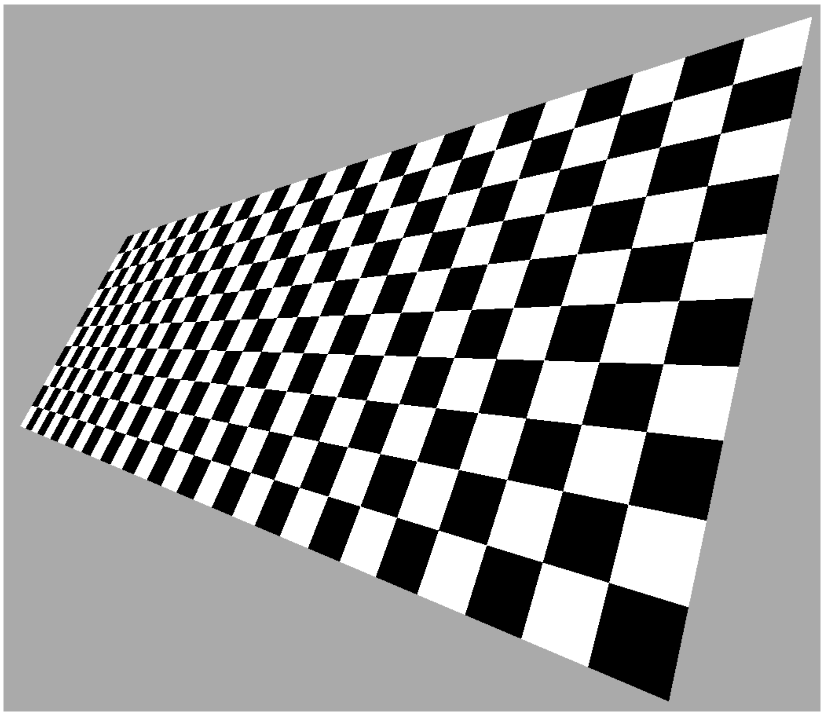

Figure 3: *Image produced by correct algorithm of* §*3.5 using rational linear interpolation of texture coordinates. Note proper foreshortening.*

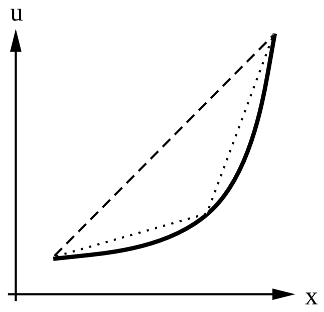

Figure 4: *An object-affine parameter u is a rational linear function of screen x. Solid curve: correct function $u(x) = (ax + b)/(cx + d)$; dashed curve: piecewise-linear approximation for one polygon; dotted curve: linear interpolation with two polygons.*

are never exact. The per-polygon and per-vertex cost of transformation, clipping, and scan conversion may even cancel the advantages of faster pixels.

3 Polygon Rendering with Rational Linear Interpolation

The "correct" solution requires *rational linear interpolation*: independent interpolation of a linear numerator and linear denominator followed by division at each pixel. In previous work, Newman and Sproull found the rational linear formula relating a linear interpolation factor (between 0 and 1) for screen space to the interpolation factor for eye space [Newman-Sproull76 p.362]. Smith applied a similar technique to texture mapping, showing that a divide was needed at each pixel [Smith80].

3.1 Rational Linear Interpolation the Hard Way

In previous work, the first author described incremental interpolation of texture coordinates with a per pixel cost of three additions, two divisions, and a texture access [Heckbert83]. Along each scanline, texture coordinate u has the form $u(x) = (ax + b)/(cx + d)$, and $v(x)$ is similar.

The method employed for computing the homogeneous texture coordinates (uq, vq, q) at each vertex was quite involved, however [Heckbert89]. First, it required inference of the affine texture-to-object parameterization from the correspondence at three vertices of the polygon. This mapping was then concatenated with the object-to-screen mapping to arrive at the 3×3 projective mapping matrix. The screen coordinates of each vertex of the clipped polygon were transformed by the inverse of this matrix to compute the homogeneous texture coordinates, which were linearly interpolated across the polygon. The cost of texture mapping setup with this method was 133 arithmetic operations (multiplies and adds) per polygon, plus 12 arithmetic operations per vertex. Transformation and clipping are usually done in floating point, but scan conversion can work in 32-bit integer arithmetic if done carefully.

3.2 The Easy Way: Derivation

There is a much simpler alternative, however! As observed by the second author, the homogeneous texture coordinates suitable for linear interpolation in screen space can be computed simply by dividing the texture coordinates by screen w, linearly interpolating $(u/w, v/w, 1/w)$, and dividing the quantities u/w and v/w by $1/w$ at each pixel to recover the texture coordinates. Any object-affine parameter may be interpolated in this fashion. To demonstrate this, we need the following theorem:

Theorem:
Given:

1) n parameters $r_1, r_2, \cdots, r_n$ that are object-affine,

2) a 3-D object space that is 3-D-screen-projective, and

3) a plane in object space that is not "edge-on" to screen space;

then the parameters are screen-projective on this plane.

Proof:

We write the homogeneous parameter space as $\mathbf{p_r} = (r_1 w_r, \cdots, r_n w_r, w_r)$, the homogeneous object space as $\mathbf{p_o} = (x_o w_o, y_o w_o, z_o w_o, w_o)$, homogeneous 3-D screen space as $\mathbf{p}_\sigma = (xw, yw, zw, w)$ and homogeneous (2-D) screen space as $\mathbf{p_s} = (xw, yw, w)$. Let $\mathbf{M_{ab}}$ denote the transform matrix from a space to b space. Then the parameters are $\mathbf{p_r} = \mathbf{p_o M_{or}}$ for some $4 \times (n+1)$ parameterization matrix $\mathbf{M_{or}}$, and since the parameters are object-affine, the last column of this matrix is $(0,0,0,1)^T$, so $w_r = w_o$. Similarly, since object space is 3-D-screen-projective, we have $\mathbf{p_o} = \mathbf{p}_\sigma \mathbf{M}_{\sigma\mathbf{o}}$ for some 4×4 matrix $\mathbf{M}_{\sigma\mathbf{o}}$.

Since projective mappings preserve planes, a plane in object space transforms to a plane in screen space, and since the plane is not edge-on, the plane has a unique depth z at each (x, y), so $z = \alpha x + \beta y + \gamma$ for some α, β, and γ. On this plane,

$$\begin{pmatrix} x & y & z & 1 \end{pmatrix} = \begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & \alpha & 0 \\ 0 & 1 & \beta & 0 \\ 0 & 0 & \gamma & 1 \end{pmatrix}$$

or $\mathbf{p}_\sigma = \mathbf{p_s} \mathbf{M}_{\mathbf{s}\sigma}$.

Since the screen-to-3-D-screen, 3-D-screen-to-object, and object-to-parameter mappings are all projective, their composition is projective, so the screen-to-parameter mapping is projective, and $\mathbf{p_r} = \mathbf{p_s M_{sr}}$, where $\mathbf{M_{sr}}$ is a $3 \times (n+1)$ matrix:

$$\mathbf{M_{sr}} = \mathbf{M}_{\mathbf{s}\sigma} \mathbf{M}_{\sigma\mathbf{o}} \mathbf{M_{or}} = \begin{pmatrix} a_1 & a_2 & & a_n & A \\ b_1 & b_2 & \cdots & b_n & B \\ c_1 & c_2 & & c_n & C \end{pmatrix}$$

for some a_i, b_i, c_i, A, B, and C. ∎

The parameter values on the plane are thus related to the screen coordinates via: $(r_1 w_r, \cdots, r_n w_r, w_r) = (xw, yw, w)\mathbf{M_{sr}}$, so

$$(r_1 w_r/w, \cdots, r_n w_r/w, w_r/w) = (x, y, 1)\mathbf{M_{sr}} \tag{1}$$

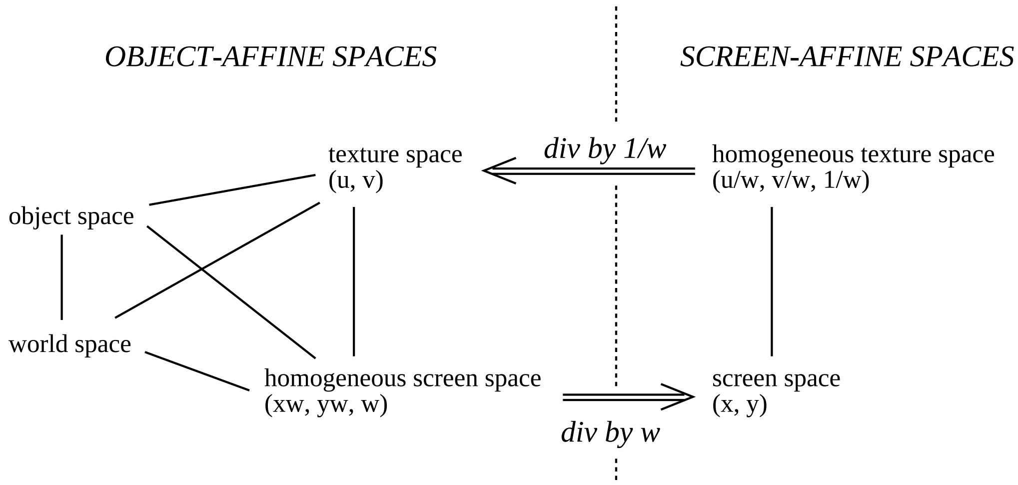

Figure 5: *Interrelationship of the coordinate systems required for standard texture mapping. The parameter set is (u,v). Affine transformations relating two spaces are indicated by a solid line, projection transformations are indicated by a double arrow.*

is screen-affine on this plane. To solve for the parameter values at each pixel we could simply compute the matrix multiply above, but that would be unnecessarily slow. A faster, incremental method is feasible if we can determine the unknown function $w_r(x,y)$.

If we start with a point on the plane that has object space coordinates $\mathbf{p_o}$ with $w_o = 1$, transform it through the mapping $\mathbf{M_{o\sigma}} = \mathbf{M_{\sigma o}}^{-1}$, and discard z, we can compute homogeneous screen coordinates (xw, yw, w). If this point is then transformed back to object space, we will recover $\mathbf{p_o}$, of course, but most importantly, its homogeneous coordinate w_o will be unchanged, since the concatenated matrices annihilate each other and the point is on the plane. Transforming further to parameter space, we find that $w_r = 1$, since $w_r = w_o$. And since we transformed all the way back from screen space to parameter space, equation (1) applies, so

$$\frac{r_i(x,y)}{w(x,y)} = a_i x + b_i y + c_i, \qquad \frac{1}{w(x,y)} = Ax + By + C$$

The homogeneous parameter space so computed is screen-affine.

3.3 The Easy Way: Summary

If the parameters are affine with respect to object space, and the homogeneous screen coordinates are computed by transforming an object space point with $w_o = 1$ (or any nonzero constant), then the homogeneous parameter vector $(r_1/w, \cdots, r_n/w, 1/w)$ is screen-affine, so it can be interpolated with linear interpolation in screen space (figure 5).

Setup for interpolation is much simpler with this method than with the previous method. None of the matrices used in the proof need to be computed, and no matrix multiply to transform from screen space to homogeneous parameter (or texture) space is needed. The only setup required is that each parameter value be divided by the screen w at that point, and an extra parameter with value $1/w$ be added to the interpolated-variable list. To compute n parameters, $n+1$ divisions are needed per vertex, and $n+1$ variables must be interpolated. At each pixel, we divide the n interpolated homogeneous parameters by the interpolated $1/w$ to compute each parameter value:

$$r_i(x,y) = \frac{r_i(x,y)/w(x,y)}{1/w(x,y)} = \frac{a_i x + b_i y + c_i}{Ax + By + C}$$

Note: on most machines the fastest way to divide n numbers by a common value is to compute the reciprocal of that value and then perform n multiplications.

If it turns out that w is identical at all the vertices, then the parameters are screen-affine for that polygon, and the division at each pixel can be avoided.

3.4 Generalizations and Limitations

Gouraud and Phong shading often require a renderer to interpolate parameters with arbitrary values at the vertices. If the interpolation method described here is used, but the first condition of this algorithm is not met, that is, the parameters are not object-affine, then for polygons with four or more sides, the results will be rotation variant, in general. Interpolation of arbitrary data over polygons with five or more sides requires mappings more complex than those discussed here, but for quadrilaterals, interpolation can be done using projective mappings. To do this we assume that the parameters are object-projective, not object-affine. Our interpolation technique can be generalized further for the interpolation of parameters all of which are screen-projective but some of which are not object-affine. For example, with a projective texture parameterization, $w_o = 1 \not\Rightarrow w_r = 1$, so we can't assume $w_r = 1$, as above, but if w_r were computed, we could linearly interpolate $(uw_r/w, \cdots, vw_r/w, w_r/w)$ to find the correct values of u and v at each pixel. A different homogeneous variable w_r/w must be interpolated for each mutually affine cluster of parameters in the parameter set. This generalization also allows perspective transformations to be used as modeling transformations in addition to camera transformations.

The new method cannot be used when there is no 3-D information, as in image warping. In that case, warp inference techniques must be used [Heckbert89].

3.5 New Algorithm

The rational linear rendering algorithm is:

(1) Associate a record containing the n parameters of interest $(r_1, r_2, \cdots, r_n)$ with each vertex of the polygon.

(2) For each vertex, transform object space coordinates to homogeneous screen space using 4×4 object to screen matrix, yielding the values (xw, yw, zw, w).

(3) Clip the polygon against plane equations for each of the six sides of the viewing frustum, linearly interpolating all the parameters when new vertices are created.

(4) At each vertex, divide the homogeneous screen coordinates, the parameters r_i, and the number 1 by w to construct the variable list $(x, y, z, s_1, s_2, \cdots, s_{n+1})$, where $s_i = r_i/w$ for $i \le n$, $s_{n+1} = 1/w$.

(5) Scan convert in screen space by linear interpolation of all parameters, at each pixel computing $r_i = s_i/s_{n+1}$ for each of the n parameters; use these values for shading.

4 Conclusions

We have presented a new method for setup of parameter interpolation on polygons viewed in perspective. The previous method, when interpolating two parameters, cost 133 arithmetic operations per polygon and 12 operations per vertex. The new method allows interpolation of any number, n, of parameters. It is simpler than the old method, it has no per-polygon overhead, and its per-vertex cost is $n+1$ divisions. The per pixel cost of the new method is identical to the old rational linear cost: n divisions per pixel. For texture mapping, the dominant cost is thus three divisions per vertex and two divisions per pixel.

The ability to cheaply and correctly interpolate parameters is useful for a number of shading parameters. This interpolation technique is most helpful for texture mapping, however, where the flaws of linear interpolation are most visible.

5 References

[Blinn-Newell76] James F. Blinn, Martin E. Newell, "Texture and Reflection in Computer Generated Images", *CACM*, vol. 19, no. 10, Oct. 1976, pp. 542-547.

[Heckbert83] Paul S. Heckbert, *Texture Mapping Polygons in Perspective*, NYIT Computer Graphics Lab, TM 13, Apr. 1983.

[Heckbert86] Paul S. Heckbert, "Survey of Texture Mapping", *IEEE Computer Graphics and Applications*, vol. 6, no. 11, Nov. 1986, pp. 56-67.

[Heckbert89] Paul S. Heckbert, *Fundamentals of Texture Mapping and Image Warping*, Master's thesis, UCB/CSD 89/516, CS Dept, UC Berkeley, May 1989.

[Heckbert90] Paul S. Heckbert, "Generic Convex Polygon Scan Conversion and Clipping", *Graphics Gems*, Andrew Glassner, ed., Academic Press, Boston, 1990.

[Maxwell46] E. A. Maxwell, *The Methods of Plane Projective Geometry, Based on the Use of General Homogeneous Coordinates*, Cambridge U. Press, London, 1946.

[Newman-Sproull79] William M. Newman, Robert F. Sproull, *Principles of Interactive Computer Graphics* (2nd ed.), McGraw-Hill, New York, 1979.

[Rogers85] David F. Rogers, *Procedural Elements for Computer Graphics*, McGraw-Hill, New York, 1985.

[**Smith80**] Alvy Ray Smith, "Incremental Rendering of Textures in Perspective", *SIGGRAPH '80 Animation Graphics seminar notes*, July 1980.

[**Wolberg90**] George Wolberg, *Digital Image Warping*, IEEE Computer Society Press, Los Alamitos, CA, 1990.