Incremental Rendering of Textures in Perspective

Alvy Ray Smith
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91103

28 October 1979

Published as Tutorial Notes at SIGGRAPH '80, Seattle, WA. This document was reentered by Alvy Ray Smith in Microsoft Word on Apr 6, 1999. Spelling and punctuation are generally preserved, but trivially minor spelling errors are corrected. Greek letters and subscripts, unavailable in the original, are used here for readability. Otherwise additions or changes made to the original are noted inside square brackets or in footnotes.

Abstract

An incremental, scanline-ordered algorithm for rendering a 2-dimensional texture into a planar convex polygon embedded in 3-space is presented. The textured polygon is projected into a viewing plane where it is seen in perspective. The method is shown to cost a divide and two lerps per pixel more than the well-known incremental algorithm for rendering an orthographic view of a texture or a perspective view of a "texture" consisting of a single color (lerp = linear interpolation). It is argued that the divide is necessary—that, for example, second-order incrementing is not good enough in general.

KEY WORDS AND PHRASES: incremental algorithm, texture mapping, perspective

CR CATEGORY: 8.2

Introduction

Rendering into two dimensions a picture which has been moved about in three and viewed in perspective is an interesting function to all image makers. For computer graphicists, this is not a conceptually difficult problem. It has been solved for years with pictures consisting of lines only [NS], in which case it can be accomplished in real time with off-the-shelf machines (e.g., the Evans & Sutherland Picture System II). Unfortunately, such machines do not yet exist for the more interesting class of full-color, texturally rich pictures. There are fast machines for transforming color pictures, but they are either one-of-a-kind and very expensive if they are powerful (e.g., space shuttle simulator at NASA Houston) or they are cheap but restricted in capability (e.g., the SqueeZoom type video machines). The purpose of this paper is to present as speedy an algorithm as is known for rendering the general class of color-textured polygons in perspective views on a general-purpose graphics machine (e.g., a cpu-framebuffer combination).

Definitions

A planar convex polygon is called a *polygon* here since no other type will be treated. The set of points on a polygon and interior to it form its *surface*. A mapping of the surface of a polygon into a set of colors is a *texture mapping*, or simply a *texture*, for the polygon. A polygon and a texture for it form a *flat*, from the painted surfaces used for sets in staged theatrical presentations.

A given flat—its polygon actually—is assumed to have passed through the following typical computer graphics processes:

(1) In homogeneous coordinate form, it has been moved about in *eye space* under any number of affine transformations—e.g., rotation, translation, scaling.
(2) A perspective transformation appropriate to the desired viewpoint has been applied. The flat is now in *perspective space*.
(3) It is clipped, if necessary, to the viewing solid.
(4) The homogeneous coordinate is divided through.
(5) A viewport mapping is applied to finish projecting the transformed polygon into 2-dimensional *screen space*.

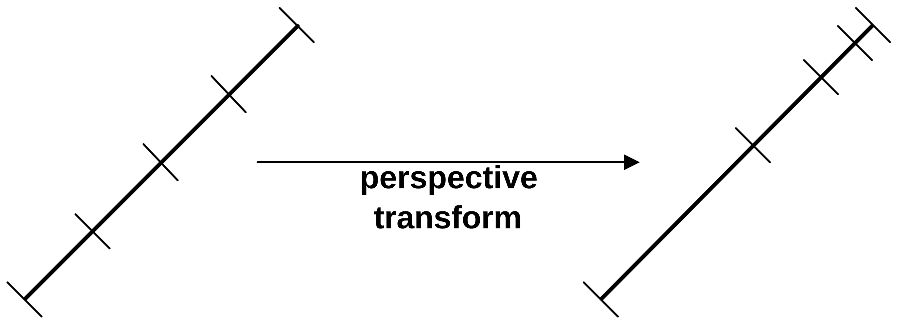
So far only the polygon has been transformed to the screen. The finished rendering requires that its texture also be transformed. If step (2) above were omitted, resulting in an orthographic projection, the following, again well-known, algorithm could be used for rendering. It is an *incremental, scanline-ordered* algorithm. It will also work for perspective projections of textured polygons where the (degenerate) texture is a solid color. Incorrectly rendered solid color looks just like correctly rendered solid color (so long as lighting is ignored).

(1) Obtain the inverse transform which takes the polygon back to its original orientation as the *source flat*.
(2) Apply this inverse transform to all vertices of the polygon to get their *inverse images*.
(3) By linear interpolation (lerping) between inverse images of adjacent vertices, determine the inverse images of the intersections of each scanline with the polygon.
(4) Again by lerping across a scanline between the inverse images computed in step (3), compute the inverse images of each pixel.
(5) Determine the color texture mapped to each pixel's inverse image and write it to the screen. This is a nontrivial step worth a paper in itself. Here we shall simply assume a routine exists for obtaining the texture mapped to the inverse image of a pixel.

The lerps in steps (3) and (4) may be computed incrementally. In step (3) Y changes by 1 for each scanline. Hence the increments of change between scanlines can be computed. Then once the inverse image of an intersection of the polygon with a scanline is known, that on an adjacent scanline is obtained by addition of the increments. In step (4) the increments of change correspond to stepping along a scanline pixel by pixel. Thus the incremental orthographic algorithm costs only two adds per target pixel.

Perspective

Unfortunately, perspective transformations invalidate the lerps in the algorithm above. See figure below.



Thus, equal increments in eye space do not map into equal increments in screen space. Below we derive a relationship between equal intervals in screen space and their nonequal inverse images in eye space. This will lead us to an incremental algorithm for rendering in perspective. The difference from the orthographic rendering above is that *two* sets of increments of change are determined for the counterparts of steps (3) and (4). Then by simple additions the numerator and denominator of a particular fraction are determined at each pixel. So the incremental perspective algorithm we derive is at least an add and a divide more expensive per pixel than the incremental orthographic algorithm. See the Appendix for an argument that the divide is necessary.

Details

The perspective transformation we shall assume here maps eye space points (X_e, Y_e, Z_e) into screen space points (X_s, Y_s, Z_s) as follows:

$$X_s = \frac{X_e}{Z_e}$$

$$Y_s = \frac{Y_e}{Z_e}$$

$$Z_s = \frac{f}{Z_e} + e$$

where

$$e = \frac{Z_f + Z_n}{Z_f - Z_n}$$

$$f = \frac{-2Z_n Z_f}{Z_f - Z_n}$$

for Z_n = near plane Z and Z_f = far plane Z, both in eye space.

Although there are several transforms that could be used, this one, derived by Ed Catmull and Jim Clark, has the nice property that X, Y, and Z (screen space) each map onto range [+1, -1]. Hence a perspective-space clipper can be written which is symmetric in all three coordinates.

We wish to determine the relationship between an interval in eye space and its image in screen space. Consider a point (X_e, Y_e, Z_e) between line segment endpoints $(X_{e_a}, Y_{e_a}, Z_{e_a})$ and $(X_{e_b}, Y_{e_b}, Z_{e_b})$. Let the images in screen space of these three points be (X_s, Y_s, Z_s), $(X_{s_a}, Y_{s_a}, Z_{s_a})$, and $(X_{s_b}, Y_{s_b}, Z_{s_b})$, respectively. Then we may say

$$Z_e = Z_{e_a} + a(Z_{e_b} - Z_{e_a})$$
$$Z_s = Z_{s_a} + b(Z_{s_b} - Z_{s_a})$$

where a and b are weights on domain [0, 1]. That is, Z_e is a lerp of Z_{e_a} to Z_{e_b} by a factor a, and Z_s is a lerp of Z_{s_a} to Z_{s_b} by a factor b. By combining the two sets of equations above, it can be shown [...1] that

(A) $$a(b) = \frac{bZ_{e_a}}{Z_{e_b} + b(Z_{e_a} - Z_{e_b})}$$

This same relationship was also derived in [NS] in a different context (hidden line removal) and using a slightly different perspective transform. The implication is, of course, that the formula for a above is independent of the actual perspective transform used.

This formula is the key to our incremental algorithm. We now derive the incremental parameters. Since we will always increment b in equal steps Δb [...2], a as a function of b for one increment is given [...3] by

$$a(b + \Delta b) = \frac{(b + \Delta b)Z_{e_a}}{Z_{e_b} + (b + \Delta b)(Z_{e_a} - Z_{e_b})}$$

or

$$a(b + \Delta b) = \frac{n + \Delta n}{d + \Delta d}$$

where,

$n = bZ_{e_a} =$ numerator of $a(b)$
$d = Z_{e_b} + b(Z_{e_a} - Z_{e_b}) =$ denominator of $a(b)$
$\Delta n = \Delta b Z_{e_a} =$ numerator increment
$\Delta d = \Delta b(Z_{e_a} - Z_{e_b}) =$ denominator increment.

Let $n_0 = b_0 Z_{e_a}$, $d_0 = b_0(Z_{e_a} - Z_{e_b})$ be the initial incremental parameters for the first b, b_0, which is in general not zero. So the Z coordinates at the endpoints of a line segment along which we wish to interpolate are used to determine n_0, d_0, Δn, Δd. With this information, we can compute the a for the *i*th increment4 of b:

$$a(i\Delta b) = \frac{n_0 + i\Delta n}{d_0 + i\Delta d}$$

So the basic routine becomes, for each scanline Y_s with X_s varying in equal increments from X_{s_a} to X_{s_b}:

1 The phrase “(using a for alpha and b for beta)” in the original is omitted here because of the availability of Greek letters.

2 The phrase “(read delta beta)” in the original is similarly omitted here.

3 The phrase “(continuing to use d for delta)” is similarly omitted.

4 The “*n*th increment” in the original is changed to the “*i*th increment” to disambiguate *n*.

(1) Compute n_0, d_0, Δn, Δd.
(2) Set $X_s = X_{s_a}$[5] and $(X_e, Y_e) = (X_{e_a}, Y_{e_a})$.
(3) Compute $\Delta X = X_{e_b} - X_{e_a}$ and $\Delta Y = Y_{e_b} - Y_{e_a}$.
(4) If $X_s = X_{s_b}$ then exit.
(5) Get the color at the inverse image of the current pixel (X_s, Y_s)—that is, at (X_e, Y_e).
(6) $n = n + \Delta n$; $d = d + \Delta d$; $a = \dfrac{n}{d}$; $X_e = X_{e_a} + a\Delta X$; $Y_e = Y_{e_a} + a\Delta Y$.
(7) goto (4)[6].

This algorithm costs two adds, one divide, and two lerps (for *X* and *Y*) per target pixel, where a lerp costs an add and a multiply. A nonincremental implementation of formula (A) costs two adds, two divides, and four lerps per target pixel.

Experience

This algorithm is a key element of the texture applying system TEXAS programmed by the author at NYIT (New York Institute of Technology) in 1978-79 [S1]. It is also an element of the scanline transformations designed by the author and Ed Catmull at JPL (Jet Propulsion Laboratory) in 1979 and reported elsewhere in this proceedings [CS].

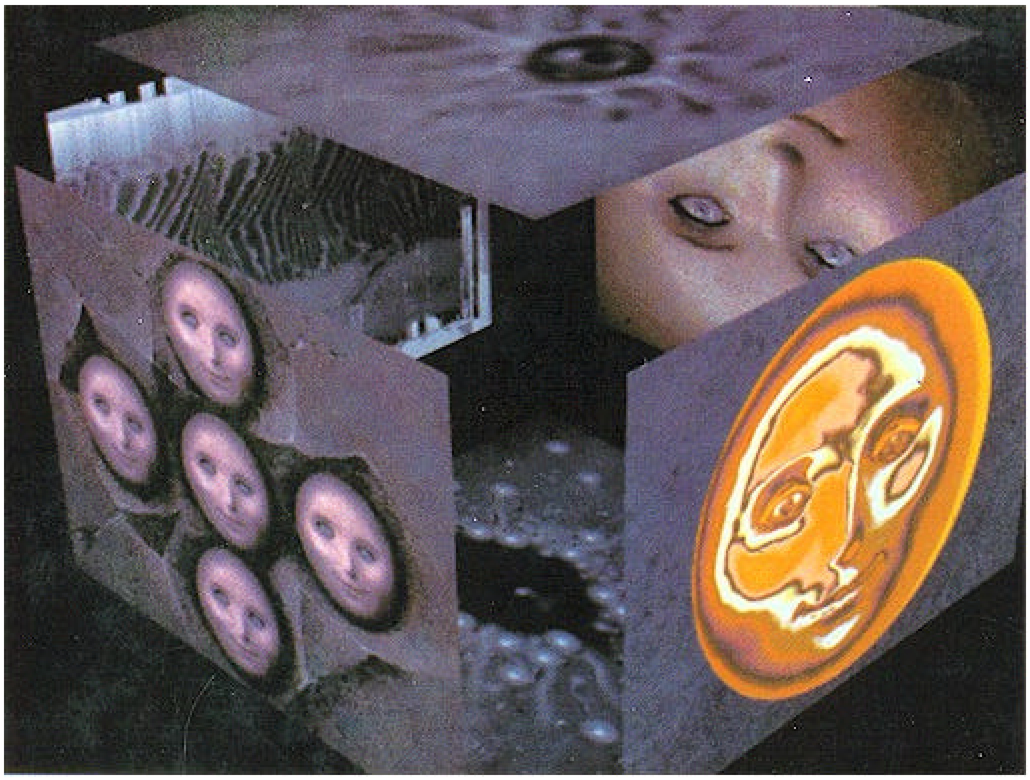


Figure 2

[5] The left side of this assignment was *X* in the original.
[6] Implicit here is that X_s is incremented by 1 at each iteration.

The object pictured in Fig. 2 was generated, using TEXAS, by mapping five paintings by Ed Emshwiller and one digitized NTSC video frame onto the faces of an exploded cube slowly rotating through space. The paintings were painted using the NYIT framebuffer paint program [S2]. The resulting picture is one frame from the video piece *Sunstone* by Ed Emshwiller (NYIT, 1979).

Acknowledgements

To Ed Catmull and Jim Clark for help and support in three dimensions, to Alexander Schure for the rich environment of the NYIT Computer Graphics Lab, and to Bob Holzman and Jim Blinn of JPL for continuing my support.

References

[1] Ed Catmull and Alvy Ray Smith, *3D Texture Mapping with Simple 2D Transformations*, submitted to SIGGRAPH ‘80, Oct 1979. [Published as *3-D Transformations of Images in Scanline Order*, SIGGRAPH ’80 Conference Proceedings, Jul 14-18, 1980, Seattle, WA, edited by James J Thomas, 279-285.]

[2] William Newman and Robert Sproull, **Principles of Interactive Graphics**, McGraw-Hill Book Company, 2nd Edition, 1979, 362.

[3] Alvy Ray Smith, *Texas (Preliminary Report)*[7], NYIT Technical Memo No. 10, Jul 1979. Also issued as tutorial notes at SIGGRAPH ‘79 (with an Addendum dated Aug 1979).

[4] Alvy Ray Smith, *Paint*[8], NYIT Technical Memo No 7, Jul 1978 (also in SIGGRAPH ’78 and SIGGRAPH ’79 Tutorial Notes, Aug 1978 and 1979 [and at SIGGRAPHs ‘80-’82 too]). [Published in: **Tutorial: Computer Graphics**, edited by John C Beatty and Kellogg S Booth, IEEE Computer Society Press, Silver Spring, MD, 2nd Edition, 1982, 501-515.]

Appendix

We shall continue to use the [notation introduced above][9] in the following Taylor series expansion of the critical formula (A) expanded about $b = 0$:

$$a(b) = b\frac{Z_{e_a}}{Z_{e_b}} \sum_{i=0}^{\infty} \left[b\left(1-\frac{Z_{e_a}}{Z_{e_b}}\right) \right]^i$$

[...[10]]. Since (Z_{e_a}/Z_{e_b}) can be anything, we cannot ignore higher order terms. This dashes any hope that there might exist higher-order difference equations for accurately approximating $a(b)$. This is true so long as no limits (other than the usual computer finiteness) are placed on the Z coordinates of the polygon being transformed.

[7] Called *TEXAS* in original paper.

[8] Called *PAINT* in original paper.

[9] Replaces “abbreviation a for alpha and b for beta” in the original, required because of lack of Greek characters there.

[10] The phrase “where SUM[] is the summation operator over the nonnegative integers n, and ** is the exponentiation operator” in the original is omitted here because of the availability of the summation and exponentiation operators. Also, i is used rather than n in the formula to avoid ambiguity. There is a missing right parenthesis in the original, just before the exponentiation operator.