IMPROVED WORLD SPACE RENDERING

Gordon C. Fossum

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-86-23 September 1986

GCF/19 September 1986

Abstract

Historically, smooth-shaded curved surfaces with proper shadows have been difficult to paint (or "render") even on powerful graphics systems. For a long time, the only methods available were referred to as "world-space" techniques, because they operated in the coordinate system where the objects of interest were described.

The advent, several years ago, of "image-space" rendering techniques, specifically "ray tracing", enabled beautiful image generation, but at great computational expense. The conceptual simplicity and high quality of ray tracing caused interest in previous methods to wane. My claim is that for some applications, ray tracing represents too much work, and that other methods can be used to generate equivalently beautiful images with much less computational expense. Specifically, I propose a novel use of z buffers and alpha buffers to generate correct curved surfaces with anti-aliased silhouettes, and correct soft shadows.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Background (or Levelsetting)

For many years, the quality of images generated by the computer graphics field has been steadily improving. However, the cost of creating these images has always been related fairly directly to the "accuracy" of the image (cost measured both in memory usage and CPU cycles; "accuracy" meaning both how "good" the image looks, and how faithful it is in representing reality.) In each of several areas of consideration, there is a spectrum ranging from blatantly unrealistic hacks to careful (and expensive) modeling techniques.

For example, there is the problem of trying to determine when a surface is visible to the eye (that is, when it is not "behind" something that is closer to the eye, at a given pixel). For a very long time, the most popular method of accomplishing this was to sort the polygons for distance to the eye, and in the case of intersecting polygons, chop them into smaller pieces until an unambiguous sort could be done. Once sorted, the polygons can be drawn in order, from furthest to closest. This method could not be easily applied to surfaces that were not polygonal (because the intersection of two non-polygonal objects was computationally difficult). An alternative method involves z buffers. A z buffer is a parcel of memory, either in general-purpose memory or in part of the frame buffer of a graphics system, which contains one or more bytes for each pixel of the image. The traditional use of a z buffer is to store, in some form, the z coordinate of the object point displayed in each pixel (hence the name). The advantage of this is that as a program prepares to paint a point on the screen, it can ask whether this point is closer to the eye than any object point that might already be displayed in the chosen pixel. If so, then paint the point and update the z-buffer value for that pixel. If not, throw the point away. This method is a marvelous solution to the hidden-surface problem.

Another area in which a whole spectrum of methodologies exists is "how to draw a non-polygonal object." The cheap way is to compute a number of points on the surface of this object, and draw the polygons that these points specify. Unfortunately, the silhouette will be polygonal, and the edges between polygons can be painfully obvious if too few points are selected. If you are selecting a color for a polygon based on its orientation vis-a-vis the light source(s), the edges can be nicely blurred by computing the average of the orientations of all polygons that meet at a given vertex, and then linearly interpolating the intensities that result from these averaged orientations across the interior of each polygon. This is known as Gouraud shading, after its inventor; see [Gour 71]. If, instead, you interpolate the orientations themselves, instead of the color intensities, (where orientations are represented by normal vectors) the resulting shadings will be more realistic and less subject to "flattening," which is the result of the maximum intensity appearing well in the interior of a polygon, and being lost because the derived intensities at the vertices are all smaller. This is "Phong shading," again, named after the inventor; see [Phon 73]. Both of these methods still suffer from polygonal silhouettes, though. The method of choice for many years for proper representation of truly curved surfaces has been the bi-cubic patch. This method was made much more computationally tractable by Edwin Catmull's subdivision algorithm [Catm 74], in which a coded representation of the 48 degrees of freedom of a bi-cubic patch can be used to generate four new sets of 48 degrees of freedom, one for each quadrant of the patch. This process can be done, he showed, using only adds and shifts (assuming integer arithmetic). Thus, it became feasible to recursively subdivide a patch until a patch piece was "down to the pixel level," at which point, it can be painted.

In most applications, it is important to be able to compute vectors from points in the scene to light sources. The cheapest hack is to place a light source at an infinite distance from the scene, allowing simplified calculation of the vector from any point in view to the light source (the same vector suffices in all cases). A more realistic approach places the light source at a finite distance away from the scene, with a concomitant increase in the cost of computing vectors from each scene point to the light source. If the light source is actually part of the scene, additional complications arise.

Determining what points are shadowed is a source of much research inquiry. In the realm of polygonal scenes, one method in wide use is "shadow volumes" [Crow 77], in which we calculate the actual volumes in which any point cannot be seen from the light source. Another method, pioneered by Lance Williams, uses z buffers for shadow computations [Will 78]. If you scan a scene with a light source placed at the eye, not painting pixels, just filling

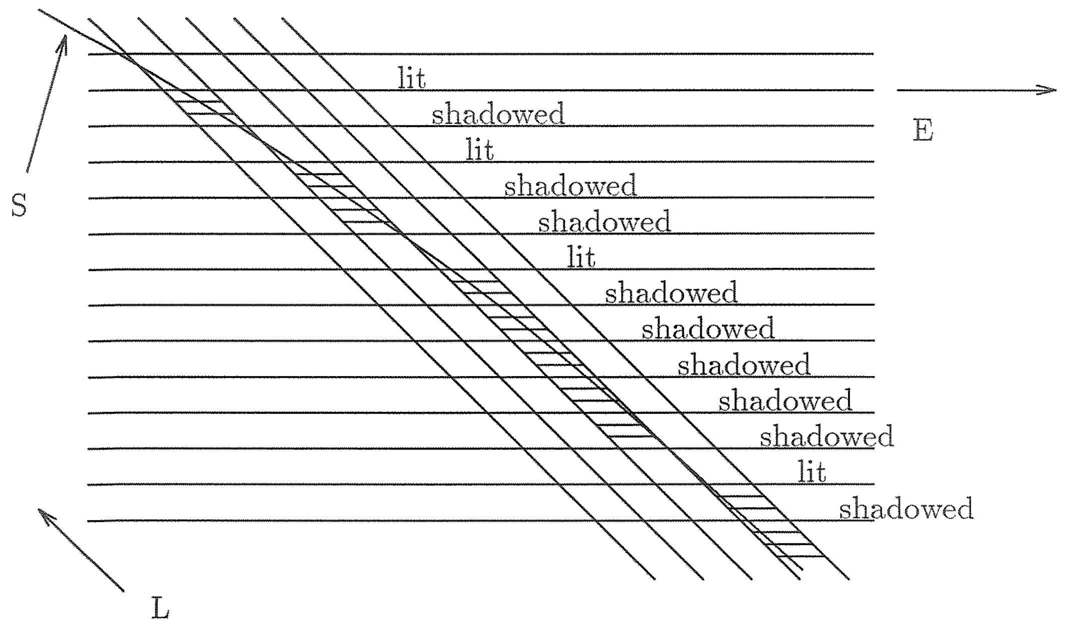

Figure 1.1: A Column of Pixels that Exhibit Self-Shadowing

a z buffer, then the resulting buffer can be used to determine which points in a scene are visible to that light source. Several drawbacks exist. One is that you need a separate z buffer for every light source. Another is that the lights are inherently point-sources. A third disadvantage is that care must be exercised to prevent the phenomenon known as self-shadowing, in which several adjacent pixels as seen from the eye and painted on the screen all belong to the same pixel in the light-source buffer. This results whenever the normal to a surface is nearly parallel to the vector to the eye, but nearly perpendicular to the vector to the light source (see figure). Here "E" shows the direction of the eye, "L" shows the direction to the light source, and "S" is the surface being rendered. In this circumstance, the classical z-buffer shadow algorithm will light only one of these visible pixels in each light-source (x,y) and shadow all others. This is pretty bad. The "fix" for this is to insist that a bias be added to the comparison which says "am I closer to the eye than the buffer value?" This bias has been represented as a fixed constant [Will 78], with the unfortunate result that no matter what the value is, self-shadowing will still occur as you get close enough to the terminator (the sunset line, if you will), and if the number is chosen too large, then an object that should be shadowed by another object close by may be accidentally rendered as lit.

Consideration of proper shadow effects leads us to consider the light source itself. The easiest model is the "point light source," but the shadows that it casts are sharp, having no penumbrae. This effect almost never "looks good", and rarely represents reality. Non-point light sources can be modeled only with considerably increased cost.

An area which causes great increase in cost, and has been the focus of much research in recent years is "how to model reflective, transparent, and/or translucent objects." Success in this area has been largely due to ray tracing. In ray tracing, the basic idea is to send a ray from the location of the eye through each and every pixel on the graphics screen (said screen being located in world-space somewhere between the eye and the scene being viewed) and on into the scene. If the ray never hits anything, then the pixel can be painted black (or whatever background color you like). When a ray hits an object, however, it immediately spawns two children: the reflected ray, and the refracted ray. For opaque objects, the refracted ray is ignored, but careful calculations on non-opaque objects lead to realistic images, albeit at the cost of maintaining a tree of spawned rays cascading from the original ray through each pixel. One flaw is that the reflected ray is usually computed to bounce in the direction it would go if the object in question were mirror-smooth. Thus, classical ray tracing does not model effects such as a spotlight generating a bright area on a piece of paper, "color-bleeding" from a non-shiny surface painted bright red to a nearby non-shiny surface painted white, or soft shadows from non-point light sources. An attempt to model these effects within the scheme of ray tracing involves shooting several rays, each equipped with access to a random number generator, to decide exactly where each ray will go [Cook 84]. This is known as "distributed ray tracing." As one might expect, this drastically increases the amount of time required to create an image.

In the last couple years, the new buzzword "radiosity" began to appear in the graphics world [Cohe 86,Cohe 85,Gora 84,Imme 86]. Radiosity refers to a method in which the output light energy of each surface in the scene is made a function of its input energy plus the light it generates on its own (the latter for light sources only). Furthermore, the direction the output energy takes is a function of the characteristics of the surface, and the directions of any input energies. So, for example, some surfaces might have equal energy output in all directions (these being known as diffuse surfaces, or matte finishes), while others really do reflect like mirrors, giving the output intensity function a distinctly spiky appearance. Most surfaces end up being a combination of the two. The radiosity model proceeds to consider the "form factor" between any two surfaces in a scene. This is the percentage of output energy from surface one that impinges on surface two. This method is expensive, but has the benefit of greater flexibility than classical ray tracing. The most complete work to date in the area of properly rendering an object, which takes into account both the effects of specular (mirrorlike) and diffuse reflection, is [Kaji 86].

1.2 Scope of Present Research

This thumbnail sketch represents the status of the subfield of computer graphics that specializes in realistic image generation. I include this introduction in order to allow me to describe where my research area fits in. I have been deeply exploring an area that was investigated in the mid-1970's, but was, for the most part, placed on the "back burner", so to speak, upon the advent of ray tracing. I refer to the use of z buffers for shadow computations.

My first minor result is to insist that the bias to control self-shadowing, which I referred to earlier, should be a function of the angle between the normal to the surface and the vector to the light source. As this angle approaches 90 degrees, increase the bias, proportionally with the tangent of the angle, up to some maximum cutoff value.

My major result is to expand on the idea of z buffers for shadow computations. I use another pre-existing application of the z buffer in a new way. This application is called the alpha buffer. An alpha buffer stores, for each pixel, the percentage of that pixel which is covered by the rendered image [Fium 83,Port 84,Carp 84]. This information is useful as an anti-aliasing tool, wherein you attenuate the intensity of the color with which you paint a pixel to account for its alpha buffer value. I propose to use a combination of two z-buffers and two alpha buffers to generate antialiased images with soft shadows, which are correct and realistic.

The important thing is to note that what we are trying to do is determine, with some measure of accuracy, how much of the light source is visible to the point that we are trying to render. If the answer is none, the object is in full shadow. If the answer is greater than none and less than all, the object is in the penumbra of the shadow. The difficulty is in ascertaining this value.

The first step is to render the scene from the point of view of the light source, taking care to note, for each pixel, the distance to the closest object point, the distance to the next-closest object point, and the percentage of the pixel covered by the closest object point. The next step is to consider each of the two points represented by the two z-values. For some neighborhood around the pixel, see if anyone else's points come between you and the light source (noting here that the light source is NOT a point source, and that its shape is determined by some small, simple formula, as for a sphere, a rectangle, or a line segment). If such a partial eclipse occurs, the information is readily available to determine what percentage of the light source is eclipsed by that point (here visualizing the point as a small black square of the appropriate size). The result is that the alpha buffer can be converted from representing the geometrical notion of what part of a pixel is covered by an object to representing what percentage of the light source is visible to the object at that point.

This dissertation proposal explains, in detail, the operation of my test bed, and the usage of z buffers and alpha buffers to do proper shadowing. Thus far, I have successfully incorporated the alpha buffer concept into my test bed to the extent that the bi-cubic patches I render are now anti-aliased. It remains for me to incorporate the full combined z/alpha buffer concept to verify its correctness.

Chapter 2

The New Idea

2.1 Motivational Recap

The traditional use of z-buffer concepts to determine shadowing does not allow for non-point light sources. You need one z buffer (which may run to 1.5 megabytes or so) for each light emitting point that impinges on your scene. In a scene with, say, five flourescent light panels in the ceiling, you are not going to get realistic lighting effects with traditional z buffers until and unless you are willing to invest in several gigabytes of memory.

Given that memory continues to get less expensive, it is reasonable to investigate algorithms which require more memory than would have been considered wise in years past. The algorithm that I propose here will require approximately two megabytes per light source, but these will be modellable as non-point light sources.

2.2 The Basic Implementation

Consider the use of two z buffers and two alpha buffers per non-point light source. Again, the goal is to understand how much of the non-point light source is visible at a point on a surface (that is, an entry in a z buffer). Label the z buffer representing the closest thing to the light source in that pixel as the "upper" z buffer. Label the other as the "lower" z buffer. Then the "upper" alpha buffer record contains the fraction of the pixel which is covered by whatever is at the depth recorded in the upper z buffer, and the lower alpha buffer is initially empty.

Now, note that as you fill the lower and upper z buffers initially, it is trivial to store the location that is closest to the light source; the minimum z value in either buffer. Further note that if you know the "radius" of the light source, and the ratio between my distance from the minimum z and my distance to the light source, you have a nice upper bound on how many of your neighbors (the buffer locations with nearby (x,y) values) can possibly block a piece of your view of the non-point light-source. The size of the bounding neighborhood differs, depending on how close you are to the minimum z value. For a light source whose radius takes up 0.25 degrees of sky, like the sun or the moon, the neighborhood that would need to be considered might conceivably be a mere 5 x 5 block of (x,y) values with the light-source pixel under consideration as the center. Of course, a light source like a flourescent tube could (and should) be handled by a different method, taking advantage of its shape, and thus limiting in a useful way the shape of the neighborhood that needs to be considered.

The algorithm will process through the z buffers, skipping those points with "max" z values, as they are background points, not part of any object. Having established the neighborhood of (x,y) values that needs to be considered as potential shadowers, the algorithm will proceed to examine each neighbor, and, using similar triangles, determine exactly which neighboring (x,y,z) locations stored in either z buffer actually do come between your location and the non-point light source. Note that points which are on the border of the light source (as seen from the point in question) can contribute fractional values to this accumulation. The grand simplification comes from the fact that we can assume that any shadowing is done by a homogeneous piece, and that, having collected the number of interfering points (say "n"), we can take their average z value, and then compute how many points, at that average z value, would be required to obscure the light source completely (call this "m"). Then the ratio n/m will be an excellent estimate of the required attenuation of the illumination from the light source in question onto the patch location in question. This ratio is then stored in the alpha buffer at that (x,y) location. Recall that this is being done for both the "upper" and "lower" z value. The novel aspect of this is that points which are clearly invisible to the center of the light source can still be illuminated by the areas of the light source other than the center, and the effect is properly reproduced here.

The result is that now, we have a buffer which tells us, at every light-source (x,y) pixel, the required attenuation value for illumination of the first two objects "visible" to the light source at that pixel. Again, this means that stuff which is invisible to the center of the light source can still be partially illuminated. This is the essence of soft shadows.

2.3 Status, Research Direction and Discussion

Thus far, I have successfully incorporated the alpha buffer concept into my test bed to the extent that the bi-cubic patches I render are now anti-aliased. It remains for me to incorporate the full combined z/alpha buffer concept. My goal over the next year is to demonstrate the validity of this concept. I shall also measure its efficiency against appropriate existing implementations which generate soft-shadowed anti-aliased parametric patch scenes. My proposed method of comparing efficiency is discussed in the next chapter. Some detailed comments on undecided issues follow:

The simplification in the previous section need not occur. However, trying to take into account multiple occluding pieces will require some sort of data structure to accumulate how many points are occluding the light source at each level, and I do not believe that such a level of care will be useful, even in a relatively complicated scene.

The notion that only a portion of a neighbor may be present (that is, the neighbor represents a patch piece that does not cover the entire pixel at that z value) is not taken into consideration here. I trust that for most cases this nettlesome detail can be ignored with impunity, but further testing on my part will be needed to determine whether the effect is satisfactorily realistic. If not, the number of required buffers may rise from four to six: two for z values, two for coverage values and two for illumination attenuation factors.

Chapter 3

A Description of the Testbed

3.1 Overview

The basic idea is to let the user control the shape of the patch, as a wireframe diagram, by adjusting the values of any or all of the 48 degrees of freedom which are available in the specification of a b-spline bicubic patch (x, y, and z for each of sixteen points). When you have a patch that you would like to see rendered, a mouse button push starts this process.

First, the z buffers from two different light sources are computed, and then the painting process actually starts. The patch is recursively subdivided until a piece is detected which is "at the pixel level." This piece's visibility to the eye is determined (using a z buffer in the classical way), then, if visible, its color intensity is computed (using vector calculations, as well as shadow calculations, with the light-source z buffers), and it is painted on the screen. This color-intensity part is where the modified z/alpha buffer concept will come into play, when it is perfected. The two z buffers previously referenced will actually be more complex than they are right now.

On completion of the patch, the program automatically writes it to disk for future display, and offers mouse button control to leave or to repeat the process with a new wire frame.

3.2 Detail of Wire-Frame Mode

A b-spline bi-cubic patch represents the warping of a square of (u,v) parametric space (where u and v are both in the interval [0,1]) into x,y,z space, using three functions on the independent variables u and v; one function for x, one for y, and one for z. Each function has sixteen terms, one for each possible combination of a power of u multiplied by a power of v, from constant (the zero-th power) up to and including cubic terms. Thus, the sixteen coefficients of this "bi-cubic" function control where the function goes, as you range over this unit square in (u,v). With three such functions, you can see where the 48 degrees of freedom come from. The magic of all this is that a few well chosen matrices can convert the 48 numbers contained in 16 "control points" into the 48 numbers needed for these three functions. This is the wonder of b-splines. These 16 control points are the vertices of a convex polyhedron, and the patch is constrained to lie within this polyhedron.

The way I let a user control these degrees of freedom is to show 48 vertical bars (grouped into 16 clusters of x,y,z values, and all placed in the rightmost one-quarter of the screen), each of which can vary from -1.0 through +1.0. The user can choose the value of each bar with appropriate mouse buttons (see figure). This representation is not very convenient, ergonomically, but testbeds are not intended to be marketable.

The way I get a wire-frame representation of a patch is to sequentially process through a bunch of fixed values for u and v, and just draw the lines on the screen, using standard 3d transformations, from world-space coordinates. (As the system exists now, I actually draw them in stereo, so that the red-blue 3D glasses give you true depth perception, but this really is not

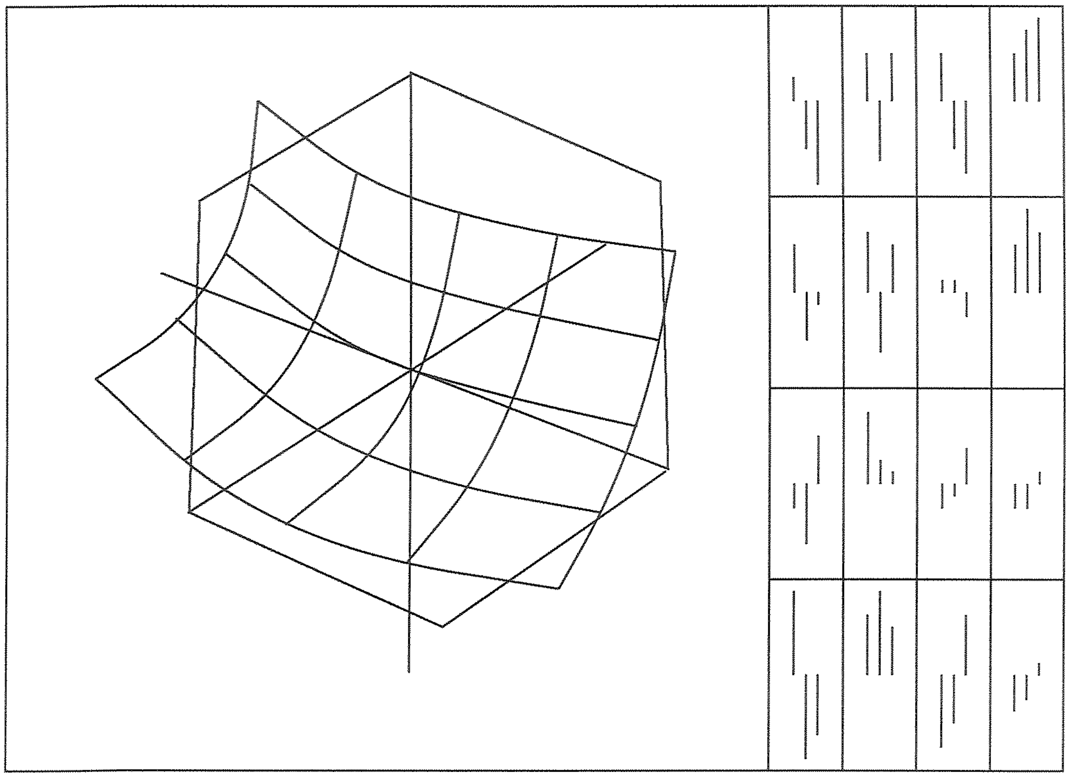

Figure 3.1: The Screen, in Wire-Frame Mode

germane to my research.) Besides changing the shape of the patch (in near-real time) by modifying one of the 48 valuators, the user can, under mouse control, change the eye position in real time. This, in conjunction with the stereo representation, permits a very strong perception that you know what the patch will look like before you commit to start painting.

3.3 Detail of Light-Source Z Buffers

The process described here is the existing version. The description of the proposed version will come in the next chapter. There are two light sources that are presently hard-coded into the program. One is located at (0.0, 17.0, 0.0) in world-space coordinates. This light is colored blue. The other is at (0.0, 0.0, 17.0) and is colored green. The program does the following for each of the two light sources: place the "eye" at the light source and set up the machine's transformation matrices accordingly; recursively subdivide the patch, and when at the pixel level, decide which pixel (in the 512 by 512 region I have presently allocated for patch display) this patch piece belongs to. If the z coordinate of this piece is further away than the z value presently stored in that (x,y) location in the buffer, do nothing. Otherwise, store the new z value in the buffer. Note that I do not compensate this value in consideration of the normal vector to the surface when I build the buffer. These calculations are done at the time that points are actually calculated for painting. (Recall my discussion of my first "minor" result, in the previous chapter.) While this buffer-building algorithm proceeds, the program displays a rudimentary counter on the screen to keep the user informed of the status. Building the buffers takes anywhere from 20 seconds to 30 minutes, depending on the size of the patch, as seen from the light sources.

3.4 Detail of the Painting Algorithm

Again, the program recursively divides the patch, and when the piece under consideration is determined to fall in an area comprising a square on the screen which is two pixels on a side, the computation begins. First, the piece's z value is compared to that currently in the viewer's z buffer. If the point is not visible, go on to another patch. Otherwise, proceed to compute how much area in each of the four pixels in the box is covered by the piece under consideration. Add these values to the alpha buffer entries for those pixels. Compute the normal to the patch. Determine if the patch is visible as seen from each light source, taking into account the variable bias discussed earlier. If so, using vectors again, determine how much blue intensity and/or how much green intensity to assign to the affected pixels (those which are partially covered by the piece in question) above the background ambient color intensities (presently a dull charcoal gray). Paint each affected pixel with the indicated intensities, attenuated by the values in the alpha buffer. When the recursive algorithm is complete, write the image to disk. The painting algorithm takes anywhere from 30 seconds (for a patch of one square centimeter on the screen) to about 40 minutes for a good-sized patch. The time required to write to disk is approximately 10 seconds.

3.5 Efficiency Measurement

There are many examples of parametric patches in the literature. There are also many examples of images with soft shadows in the literature. There are not very many examples of parametric patches with soft shadows. There may be some difficulty in finding a test image that has been previously published which has parametric patches and soft shadows; given that I find one, I will generate a similar image, and see which image takes longer (taking into consideration the relative computing power of the competing host machines), and include these results in my final dissertation. If I cannot find a suitable test image, I shall resort to a ray-traced image involving parametric patches, even though it may not have soft shadows, to see if my method is computationally competitive with this. Of course I shall have to attempt to put a cost on the additional memory I shall require, as opposed to the memory requirements of previous methods.

BIBLIOGRAPHY

[Carp 84] Loren Carpenter. *The A-buffer, an Antialiased Hidden Surface Method.* Computer Graphics, Vol 18 No 3, July 1984.

[Catm 74] Edwin Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD Dissertation, University of Utah, December 1974.

[Cohe 86] Michael Cohen, Donald Greenberg, David Immel, Philip Brock. "An Efficient Radiosity Approach for Realistic Image Synthesis," *IEEE Computer Graphics and Applications,* Vol 6 No 3, March 1986.

[Cohe 85] Michael Cohen, Donald Greenberg. *A Radiosity Solution for Complex Environments.* Computer Graphics, Vol 19 No 3, July 1985.

[Cook 84] Robert Cook, Thomas Porter, Loren Carpenter. *Distributed Ray Tracing.* Computer Graphics, Vol 18 No 3, July 1984.

[Crow 77] Franklin Crow. *Shadow Algorithms for Computer Graphics.* Computer Graphics, Vol 11 No 3, July 1978.

[Fium 83] Eugene Fiume, Alain Fournier. *A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General Purpose Ultracomputer.* Computer Graphics, Vol 17 No 3, July 1983.

[Gora 84] Cindy Goral, Kenneth Torrance, Donald Greenberg, Bennett Battaile. *Modeling the Interaction of Light Between Diffuse Surfaces.*

Computer Graphics, Vol 18 No 3, July 1984.

[Gour 71] Henri Gouraud. “Continuous Shading of Curved Surfaces,”*IEEE Transactions on Computers,* Vol 20 No 6, June 1971.

[Imme 86] David Immel, Michael Cohen, Donald Greenberg. *A Radiosity Method for Non-Diffuse Environments.* Computer Graphics, Vol 20 No 4, August 1986.

[Kaji 86] James Kajiya. *The Rendering Equation.* Computer Graphics, Vol 20 No 4, August 1986.

[Phon 73] Phong Bui Tuong. *Illumination for Computer Generated Images.* PhD Dissertation, University of Utah, July 1973.

[Port 84] Thomas Porter, Tom Duff. *Compositing Digital Images.* Computer Graphics, Vol 18 No 3, July 1984.

[Will 78] Lance Williams. *Casting Curved Shadows on Curved Surfaces.* Computer Graphics, Vol 12 No 3, August 1978.