Compositing 3-D Rendered Images

Tom Duff

Room 2C-425
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

ABSTRACT

The complexity of anti-aliased 3-D rendering systems can be controlled by using a tool-building approach like that of the UNIX™ text-processing tools. Such an approach requires a simple picture representation amenable to anti-aliasing that all rendering programs can produce, a compositing algorithm for that representation and a command language to piece together scenes. This paper advocates a representation that combines Porter and Duff's compositing algebra with a Z-buffer to provide simple anti-aliased 3-D compositing.

CR Categories and Subject Descriptors: I.3.3 [Picture and Image Generation] Display algorithms, Viewing algorithms, I.3.5 [Computational Geometry and Object Modelling] Curve, surface, solid and object representations, I.3.7 [Three-Dimensional Graphics and Realism] Visible line/surface algorithms

General Terms: Algorithms

Additional Keywords and Phrases: image synthesis, 3-D rendering, hidden-surface elimination, anti-aliasing, Z-buffer, compositing

1. Introduction

3-D rendering programs capable of dealing with detailed scenes are usually large and complex. For example, the version of REYES [1] used at Lucasfilm to create "The Adventures of André & Wally B." [12] is a 40,000 line C program. There are at most two people who understand it in its entirety.

There are several approaches to controlling this complexity. The NYIT Computer Graphics Laboratory has a set of special-purpose rendering programs that all use Z-buffer algorithms (see below) [4]. Each can initialize its Z-buffer with the results produced by the others and add objects to a scene. Thus, to produce a scene containing quadric surfaces, fractal terrain and polyhedra, three simple programs, each rendering one surface type can replace a combined quadric surface/fractal terrain/polyhedron rendering program.

Frank Crow at Ohio State University built a system that combines the output of heterogeneous rendering programs using a list-priority algorithm [3]. [15] describes a rendering test-bed that reduces objects on-the-fly to polygons, slices the polygons into spans the height of scan lines or smaller, and combines the spans using a Z-buffer (usually). The compositing algebra described in [10] is used at Lucasfilm to combine the output of many rendering tools (including REYES).

Their method is a list-priority algorithm with the list ordering worked out manually. A similar algorithm is used at most motion picture special effects houses, running on optical printers instead of digital computers [7].

All these systems have drawbacks. Z-buffer methods are hard to anti-alias, because their data representation is point-sampled (but note chapter 7 of [2]). The Lucasfilm and Ohio State systems require that all surfaces be linearly separable [9]. Even when it is possible to separate a scene by dicing it with cutting planes [5], the primitives in the diced scene may be more complex than in the original. It may even be impossible to find separating planes for scenes containing surfaces that intersect in non-planar curves.

Whitted and Weimer's approach requires that the various rendering sub-methods be connected by a complex polygon-span data structure. Considerable understanding of the system's internals is required to add new features. Anti-aliasing is difficult, but easier than with other Z-buffer style methods.

A similar complexity problem obtains with many document preparation systems and "integrated application environments." Emacs [6] is a text-editor with 450 commands, and more coming every day. Lotus 1-2-3 is a desk calculator with a 250 page instruction manual. The UNIX text processing tools [8] avoid this syndrome by cutting the text processing problem into many small sub-problems, with a small program to handle each piece. Because all the programs read and write a simple common data representation (ASCII character streams with end-of-line marked by a newline character) they can be wired together for particular applications by a simple command language. We would like to apply this philosophy to the problem of anti-aliased 3-D rendering. To do that, we need a simple picture representation that all our rendering programs can produce, a compositing algorithm for that representation and a command language with which to piece together scenes.

2. The rgbαz Representation

The **rgbαz** representation is a straightforward combination of the **rgbα** representation in [10] and a Z-buffer [2]. The hidden-surface algorithms for these representations are based on binary operators that combine a pair of images f and b pixel-by-pixel to produce a composite image $c = f$ **op** b. Applying the operator to a sequence of images in an appropriate order will produce the final image of the visible surfaces.

The Z-buffer operator f **zmin** b operates on a color value **rgb** and a depth coordinate **z** stored at each pixel in the frame buffer. The composite image has $\mathbf{rgb}_c = (\mathbf{if}\ \mathbf{z}_f < \mathbf{z}_b\ \mathbf{then}\ \mathbf{rgb}_f\ \mathbf{else}\ \mathbf{rgb}_b)$, and $\mathbf{z}_c = \min(\mathbf{z}_f, \mathbf{z}_b)$ at each pixel. [13] categorizes this algorithm, along with the ray-tracing approach advocated by [14], as "brute-force image space" methods (although ray-tracing is usually done in object space), dismissing both as impractical. Ironically, they are the two most popular hidden surface algorithms in use today.

The **rgbα** compositing operator f **over** b operates on pixels containing

an **rgb** value and a value α between 0 and 1, which may be thought of as the fraction of the pixel that the object covers. Each component of **rgb** is between 0 and α (see [10] for details).

The operator **over** computes $\mathbf{rgb}_c = \mathbf{rgb}_f + (1-\alpha_f)\mathbf{rgb}_b$ and $\alpha_c = \alpha_f + (1-\alpha_f)\alpha_b$ at each pixel. The foreground $\mathbf{rgb}_f$ is unattenuated at each pixel, and $\mathbf{rgb}_b$ shows through more as α_f decreases. When $\alpha_f = 1$, $c = f$ and when $\alpha_f = 0$, $c = b$, since each component of $\mathbf{rgb}_f$ must be 0 when $\alpha_f = 0$. Using **over** with more than two elements requires knowledge of their front-to-back order, so that the operator can be applied to elements or previous composites that are adjacent in depth.

f **over** *b* is inherently anti-aliased (really area-sampled) if, as is usually the case, α_f and α_b are uncorrelated. It can make mistakes when, for example, two elements share an edge. The only apparent way to solve this problem is to store an unbounded amount of information inside each pixel. [1] is an example of such a method, but it runs slowly and is difficult to implement correctly.

zmin is commutative and associative; that is, the order in which objects are composed is irrelevant. Because the method is point-sampled, anti-aliasing is difficult. **over** trades commutativity for anti-aliasing. It doesn't care whether elements are composited front-to-back or back-to-front or some recursive combination of the two, but they must be adjacent in depth when they are combined.

The **rgbαz** algorithm's compositing operator **comp** combines the action of **zmin** and **over**. Each pixel contains **rgb** and α along with the **z** value at each corner. Corners that are not covered have **z** set to a value (called $+\infty$) larger than any legitimate **z** value. Since each **z** value is used in 4 pixels, we keep the upper left-hand corner **z** with its pixel and get the other values from adjacent pixels. (This means that we must store an extra column off-screen at the right, and a row off the bottom, whose **rgb** and α we never use.)

f **comp** *b* is computed by first comparing $\mathbf{z}_f$ to $\mathbf{z}_b$ at each corner of the pixel. There are $2^4 = 16$ possible outcomes. If the comparisons are not the same at all four corners, we say the pixel is *confused*. Along each pixel edge at whose ends the **z**'s compare differently, we linearly interpolate the **z**'s and find the point at which they are equal. Figure 1 shows how to divide the pixel in each case to compute the fraction β on which *f* is in front. Then $\mathbf{rgb}\alpha_c = \beta(f \textbf{ over } b) + (1-\beta)(b \textbf{ over } f)$, and $\mathbf{z}_c = \min(\mathbf{z}_f, \mathbf{z}_b)$.

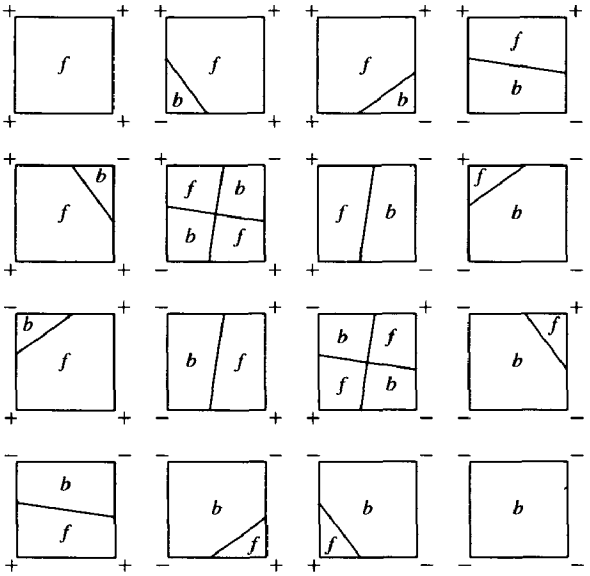

Figure 1 – The 16 Cases

Each square represents a pixel. The corners are marked with the sign of $\mathbf{z}_b - \mathbf{z}_f$. The label in each pixel fragment indicates which picture is visible in it. β is the total area of the fragments labeled *f*.

If $\mathbf{z}_f$ or $\mathbf{z}_b$ is $+\infty$ at some corner, we must pick an appropriate value to use as a surrogate in the comparisons and interpolations above. Currently, we circle the pixel clockwise and anticlockwise from the uncovered corner looking for two legitimate **z**'s and use their average.

Note that **comp** is commutative, since if we interchange *f* and *b*, β is replaced by $1-\beta$. For unconfused pixels, the result's **rgbα** is just *f* **over** *b* or *b* **over** *f*, as appropriate, and therefore the operation is associative in the unconfused case. This is almost good enough to let us combine pictures in any order. Since linearly separable objects have no confused pixels, **comp** performs just as well on them as **over**. For confused pixels associativity breaks down and mistakes can occur. For objects that legitimately intersect, the algorithm effectively computes a sub-pixel resolution polygonal curve that will be close to the intersection curve of the original objects and correctly mattes the objects together along the approximate curve. If an element *q* is between *p* and *r*, the pixels of *p* **comp** *r* can be confused with those of *q*, so elements cannot be composited in completely arbitrary order. Fortunately, the errors are usually small, so a cavalier attitude to compositing order is at least partially justified. If these errors are a problem, a group of elements can be composited all at once by sorting their pixels on their **z** coordinates and applying **comp** on adjacent elements. Then confusion-induced errors will only arise among elements whose confusion is intrinsic.

The **comp** algorithm may also encounter errors caused by the point-sampled **z** coordinates. In particular, small objects may be lost if they fall between the pixels. Furthermore, pixels are combined by area-sampling rather than convolution with a higher-order filter, which can introduce slightly scalloped intersection edges. **comp** shares with **over** the problem that errors can occur when its operands are not uncorrelated, as can happen in pixels crossed by many edges.

3. Examples

I have written a small set of programs to test these ideas. *3matte* executes the **comp** algorithm on a set of input pictures, producing an **rgbαz** output picture. *Quad* draws an anti-aliased rendition of a single quadric surface (figure 2), given the equation of its quadratic form. Using *3matte* to combine the output of multiple *quad* runs generates more complex surfaces, like the knobby-kneed robot of figure 3.

The *quad-3matte* combination is hardly a practical quadric-surface rendering system. It does show that with powerful compositing methods high quality renderings of significant objects can be produced using minimal tools.

Terrain generates anti-aliased perspective views of terrain from National Cartographic Information Center digital terrain data. Figure 4 is a view of central New Jersey, with the elevations exaggerated by a factor of 20. *Bg* generates background cards given the colors at the top and bottom of the screen. Figure 5 shows a small covey of flying saucers over New Jersey with a sky-colored background.

Figure 5 shows the sort of error that can be made when confused pixels are treated naively. Where a saucer passes behind the the rightmost foreground hill the silhouette of the hill is a little too dark near the peak.

Programs that do certain kinds of 2- and 3-dimensional image processing can operate on **rgbαz** pictures. For example, *hrot* rotates the hue of a picture, leaving alone its saturation and value [11]. The middle flying saucer of figure 5 was generated in the same colors as the one on the left, and had its hue rotated 30 degrees. *Fog* is a program that makes foggy images by mixing its input image with a fog color in an amount that depends on the **z** coordinates of the pixels. Figure 6 is the result of applying a purple haze to figure 5. The shadow-generation algorithm of [16] works on **rgbαz** pictures and could be enhanced to take advantage of the increased information available in the **rgbαz** representation.

These programs are all small and simple to write. *3matte* is 270 lines of C, *quad* is 428 lines, *terrain* is 339 lines, *bg* is 58 lines, *hrot* is 76 lines, and *fog* is 73 lines, for a total of 1244 lines.

Figure 5 is a frame from a short animated sequence produced using these programs. The sequence was 227 frames long, and took 34

hours and 26 minutes to compute (total wall-clock time, half on each of 2 VAX 11/750s, one with a floating-point accelerator and one without), or roughly 9 minutes per frame. Profiling the programs revealed that roughly 80% of their time was spent encoding and decoding the run-length encoded disk files in which the elements were stored. Not counting picture I/O, the time per frame was about 1.8 minutes. An extremely large frame buffer could eliminate picture I/O altogether.

4. Conclusions

The UNIX text-processing environment demonstrates that a suite of small programs acting on a common data representation, and bound together by a powerful command language, can be more powerful than a large integrated application program that tries to cover all eventualities. We believe that the same principle applies to image synthesis.

Experimentation is encouraged in an environment where little changes don't involve digging into, and possibly breaking, a huge monolithic program. New methods are easier to try out, and the consequences of failure are localized.

The **rgbαz** representation is easily produced by almost any rendering program and has a simple, fast anti-aliased compositing operation whose output is of high enough quality for all but the most exacting applications. The representation is the basis for a 3-D image synthesis toolkit. We expect that as we build more image synthesis and processing tools a rich 3-D graphics environment will emerge.

Acknowledgements

Don Mitchell designed the robot of figure 3. Rob Pike gave the paper a good critical reading.

References

[1] Loren Carpenter, "The A-buffer, an Antialiased Hidden Surface Method," *Computer Graphics*, Vol. 18, No. 3 (1984), pp. 103-108

[2] Edwin Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, Ph.D. dissertation, Department of Computer Science, University of Utah, Salt Lake City, December 1974

[3] Frank Crow, "A More Flexible Image Generation Environment," *Computer Graphics*, Vol. 16, No. 3 (1982), pp. 9-18

[4] Tom Duff, *The Soid and Roid Manual*, NYIT Computer Graphics Laboratory internal memorandum, 1980

[5] Henry Fuchs, Zvi M. Kedem and Bruce F. Naylor, "On Visible Surface Generation By A Priori Tree Structures," *Computer Graphics*, Vol. 14, No. 3 (1980), pp 124-133

[6] James Gosling *UNIX Emacs*, CMU internal memorandum, August, 1982

[7] L. Bernard Happé, *Basic Motion Picture Technology*, Hastings House, New York, 1975

[8] Brian Kernighan and Rob Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs NJ, 1984

[9] Martin E. Newell, "The Utilization of Procedure Models in Digital Image Synthesis," *University of Utah Computer Science Department*, UTEC-CSc-76-218, Summer 1975

[10] Thomas Porter and Tom Duff, "Compositing Digital Images," *Computer Graphics*, Vol 18, No. 3 (1984), pp. 253-259

[11] Alvy Ray Smith, "Color Gamut Transform Pairs," *Computer Graphics*, Vol 12, No. 3 (1978), pp 12-19

[12] Alvy Ray Smith, Loren Carpenter, Ed Catmull, Rob Cook, Tom Duff, Craig Good, John Lasseter, Eben Ostby, William Reeves, and David Salesin, "The Adventures of André & Wally B.," created by the Lucasfilm Computer Graphics Project. July 1984.

[13] Ivan Sutherland, Robert Sproull and R. A. Schumaker, "A Characterization of Ten Hidden Surface Algorithms," *Computing Surveys*, Vol. 6, No. 1 (1974), pp. 1-55

[14] Turner Whitted, "An Improved Illumination Model for Shaded Display," *Comm. ACM*, Vol. 23, No. 6 (June 1980), 343-349

[15] Turner Whitted and David Weimer, "A Software Test-Bed for the Development of 3-D Raster Graphics Systems," *Computer Graphics*, Vol. 15, No. 3 (1981), pp. 271-277

[16] Lance Williams, "Casting Curved Shadows on Curved Surfaces," *Computer Graphics*, Vol. 12, No. 3 (1978), pp. 270-274

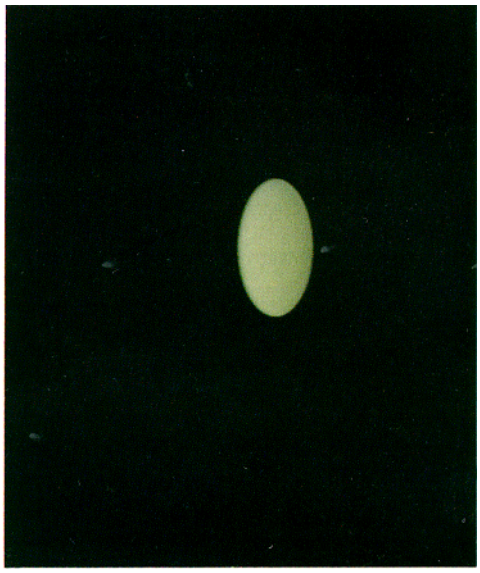

Figure 2 – Ellipsoid rendered by *quad*

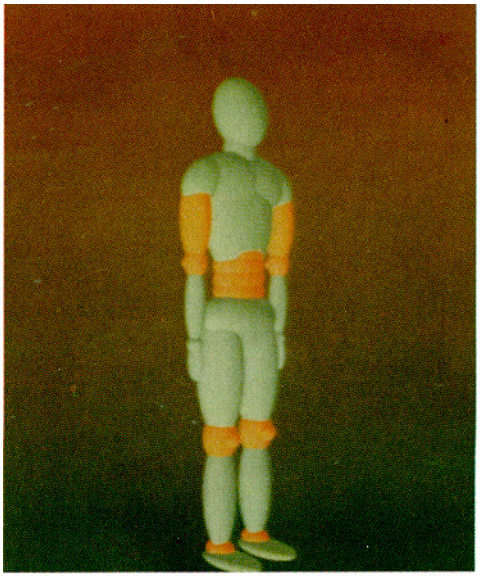

Figure 3 – Robot rendered by *quad*, *bg* and *3matte*

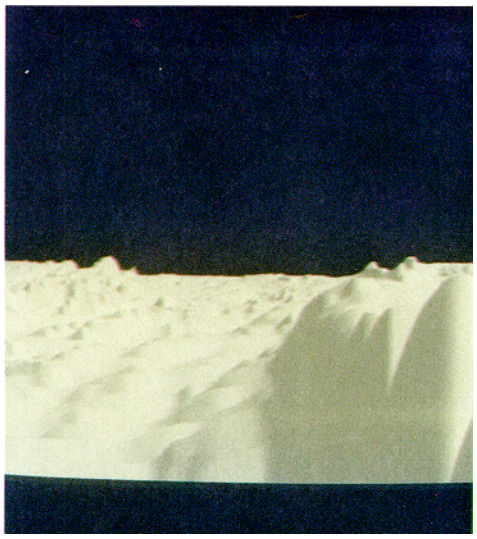

Figure 4 – New Jersey rendered by *terrain*

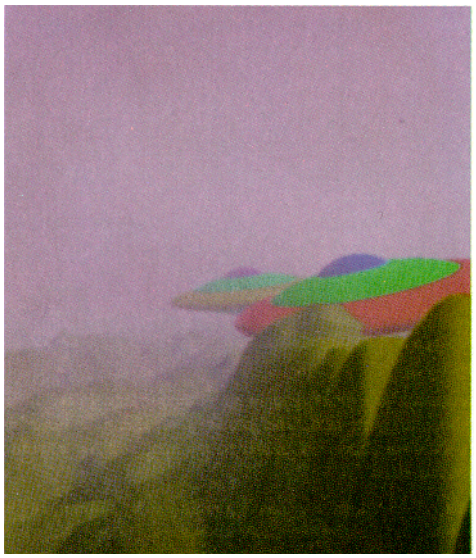

Figure 6 – Fog added to previous picture using *fog*

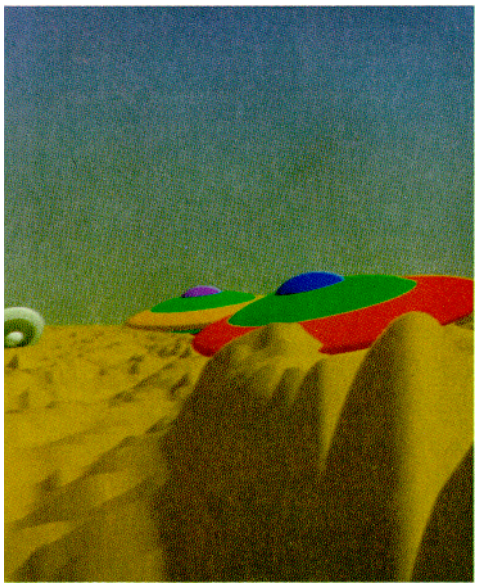

Figure 5 – Flying Saucers over New Jersey, rendered by *quad*, *bg*, *terrain* and *3matte*