A THEORETICAL AND EMPIRICAL ANALYSIS OF COHERENT RAY-TRACING

L. Richard Speer
Tony D. DeRose
Brian A. Barsky

Berkeley Computer Graphics Laboratory
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720
U.S.A.

Abstract

The use of *coherence* has been advocated as a means of reducing the large computational cost of the ray-tracing method of image synthesis. This paper examines the theoretical and empirical performance of a typical coherent ray-tracing algorithm, one that exploits the similarity between the intersection trees generated by successive rays. It is shown that despite the large degree of coherence present in a scene, the need to ensure the validity of ray-object intersections prevents any significant computational savings. This indicates that other algorithmic methods must be used in order to substantially reduce the computational cost of ray-traced imagery.

Résumé

L'utilisation de la *cohérence* a été proposée afin de réduire le coût élevé de la méthode de synthèse d'image basée sur le traçage de rayons lumineux. Cet article examine la performance, tant d'un point de vue théorique qu'empirique, d'un algorithme typique qui met de cohérence de rayons, c'est-à-dire un algorithme qui exploit la ressemblance entre les arbres d'intersections générés par des rayons successifs. Nous montrons qu'en dépit du degré élevé de cohérence présent dans une image, l'obligation de maintenir la validité des structure d'intersection fait obstacle à l'obtention de gains importants. Ces résultats donnent à penser que des méthodes algorithmiques plus fondamentales sont nécessaires pour réduire de façon substantielle les coûts de calcul du traçage de rayons lumineux.

KEYWORDS: ray-tracing, coherence.

This work was supported in part by the Defense Advanced Research Projects Agency under contract number N00039-82-C-0235, the National Science Foundation under grant number ECS-8204381, and the State of California under a Microelectronics Innovation and Computer Research Opportunities grant.

1. Introduction

The technique of tracing rays through a scene ("ray-tracing"), first used to generate shadows[1] and solve the hidden-surface problem for quadrics,[10] has become the centre of a great deal of research activity. Beginning with two papers on realistic image generation,[16,25] the method has been applied to algebraic surface rendering[12,15] and a number of problems in solids modelling.[2,17,19] Probably the most striking application of the method, however, remains its use in generating highly realistic imagery.[11,14,25] The ray-tracing method is unique in its ability to compute inter-object reflections, shadows, and accurate refraction, features that are difficult or impossible to achieve with other techniques.

The price of such effects, however, is not small. Computation times for ray-traced pictures, for example, are often measured in CPU-hours.[11,24,25] The chief reason for this is that a very large number of rays (250,000 - 1,000,000 or more) must be traced for high-quality imagery.

One strategy for reducing ray-tracing computation time relies on hardware. A number of papers have been published in this area, including one on the use of a "supercomputer",[18] two on co-processor designs[3,4] and several on multiprocessor-based systems.[5,6,23]

A second strategy uses algorithms that adaptively subdivide scenes into a number of sub-volumes. The resulting sub-volumes may or may not be disjoint from one another. In one study,[7] the subdivision is "cellular"; that is, all sub-volumes are disjoint, although together they contain the entire volume of the scene. Other papers[9,14,20] present algorithms based on a hierarchical scene subdivision. Such a scene structure generally permits fast determination of the nearest object intersected by a ray.

Another technique which has been mentioned by several authors[13,15,25] but not tried, centres on the use of *ray coherence*. As Heckbert[13] noted, "...in

many scenes, groups of rays follow virtually the same path from the eye..." (see Figure 1). As a result, the tree-like paths that are traced through the scene by successive rays from the viewpoint are often very similar. This similarity can be used to predict the path of any such ray, given the path of its predecessor,

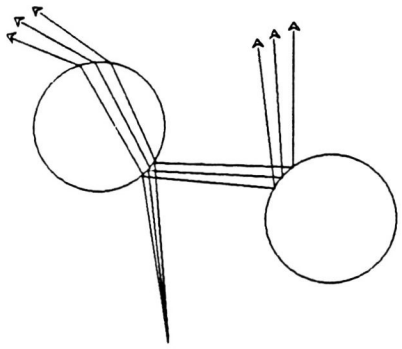

Figure 1 - Groups of Rays Follow Similar Paths

as follows. First, the ray currently being traced from the viewpoint is checked against the object intersected by the previous ray from the viewpoint. If the current ray does not intersect that object, it must be checked against all other objects in the scene, as in standard ray-tracing.[25] Otherwise, a check of the other objects need not be done, resulting in a computational savings. In this case, we say that the two rays in question are *coherent*. Also in this case, we can apply the idea recursively: any reflective ray that results is checked against the object hit by the reflective ray of the last ray from the viewpoint; and so on. It should be apparent that the degree of similarity of these paths indicates (roughly) the computational savings that coherence can provide over standard ray-tracing.

Before proceeding any further, we must add that the detection of coherence is not quite as simple as just described. Even when *corresponding* rays (two rays at the same recursion level, one of which is either the last ray from the viewpoint or one of its children, the other the current ray from the viewpoint, or one of its children) intersect the same object, the current ray might also intersect a nearer, intervening object just missed by its correspondent (Figure 2). These false-coherence cases must be detected, to produce correct results.

In the rest of this paper we describe, analyze and present performance data of an algorithm for coherent

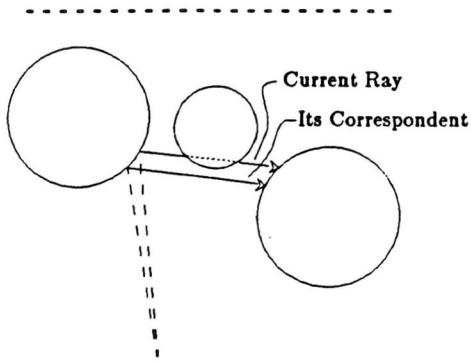

Figure 2 - Current Ray Intersects Object Hit by Its Correspondent, and Interceding Object

ray-tracing. Coherence ("the extent to which the environment or the picture of it is locally constant")[22] has often been used in graphic algorithms.[21,22] It is natural to expect coherent ray-tracing to yield the same kinds of benefits seen in other rendering algorithms. Empirically, however, we have found that this is not the case. The following sections explain why.

We should note that the algorithm we will be discusssing is most naturally used on scenes containing objects enclosed by spherical bounding volumes. The general approach, however, could also be used with other kinds of volumes.[19]

In Section 2, the coherent algorithm is presented and compared with the standard one. Section 3 gives a probabilistic analysis of the new algorithm's performance. Finally, in Section 4 we discuss our implementation and give statistics from test pictures we made.

2. Ray-tracing Algorithms

2.1. Terminology

We begin by defining terms that will be used throughout the rest of the paper. A ray is specified by an *anchor* and a *direction vector*. The anchor is the three-dimensional location of the origin of the ray. The direction vector specifies the direction of propagation of the ray.

The *image plane* is a rectangular region positioned (conceptually) in or near the scene. Elemental regions in the plane correspond to pixels of the frame buffer. Along with the plane, a *viewpoint* is specified. To simplify the discussion, we distinguish rays that originate at the viewpoint from all others

and call them *initial rays*. A *ray-set* is composed of an initial ray and any reflected and refracted rays that it generates, together with their descendants.

As mentioned in Section 1, the scene consists of some number of spherical *bounding volumes*. It will aid the discussion if we assume that relatively few of these volumes intersect, although this is not required by the new algorithm. The volumes surround *objects* composed of primitive geometric elements such as triangles or more general polygons. We refer to the latter simply as *primitives*.

Since most rays that are traced are the result of an intersection with an object, we say that a ray has an associated *originating object*. For simplicity, we will regard even initial rays as having such objects.

We now define some terms that describe the relations between a given ray and the bounding volumes present in a region or scene. Every ray naturally divides space into two half-spaces, the boundary being a particular plane (Figure 3). This plane, which we call the *bounding plane* for the ray, is defined as the plane passing through ray's anchor, having the direction vector of the ray as its normal. We call the half-space in the direction of ray propagation the *front half-space* of the given ray, and the other the *rear half-space*. A bounding volume that lies entirely in the rear half-space is said to lie "behind" the ray associated with the plane; those that do not are said to lie "in front" of the ray. We refer to the process of classifying all the volumes like this as *partitioning the region*.

Finding the intersection of a given ray and the primitives in a region requires determining which bounding volumes are intersected by the ray. All the volumes in a region that are in front of a ray divide into two groups, those that are actually intersected by the ray and those that are not. And any volume behind a ray cannot be intersected by it. Therefore, we note that a ray divides all the volumes in a region into three disjoint sets: those behind the ray, those in front but not intersected, and those in front and intersected. We call these sets B, N and I, respectively.

A few other terms that are more easily defined in context are presented later.

2.2. Standard Algorithm

We now consider the standard ray-tracing algorithm. In its simplest form, a ray-tracing program consists of two nested loops surrounding a call to a ray-tracing routine. The loops serve to scan the rows and columns of the image.

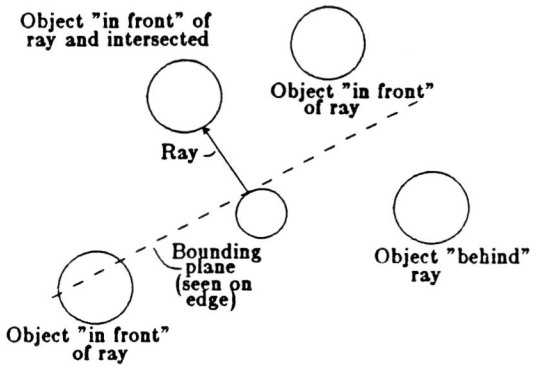

Figure 3 - Partition of Space by a Ray (Cross-sectional View)

The ray-tracing routine itself is usually written recursively. If the initial ray passed into the routine intersects an object, calls may in turn be made in the directions of reflection and refraction. In the standard algorithm,[25] recursion terminates when a ray either does not intersect anything or intersects an opaque, non-reflective surface. More recent papers[8,11] have noted that recursion can also terminate when a special attenuated-intensity coefficient that is associated with a ray drops below a threshold.

As the recursion proceeds, a record is kept of the objects intersected. Due to the fact that in many implementations, no more than two rays are spawned at an intersection, the record is often kept in the form of a binary tree (the "intersection tree" referred to in Section 1). When recursion has stopped for all members of the ray-set, the pixel's colour is computed by applying an appropriate shading rule[11,25] to the tree. After shading, the tree is discarded and the entire process repeated for the next pixel.

In addition to those traced recursively, rays are also traced from intersection points toward each light source, to test for shadowing. The results of this test are stored in the nodes of the intersection tree and used in the shading calculation.[25]

2.3. Coherent Algorithm

2.3.1. Containers Around Rays

As discussed in Section 1, a coherent ray-tracing algorithm can use the path generated by the last ray-set to predict the path of the current ray-set. Thus, one immediate difference between the new coherent

algorithm and the standard one is that after colour computation, the intersection tree is *retained* for one more program iteration. It is used during this iteration as a guide, providing hints as to which objects will be intersected by rays in the current ray-set. When tracing for the current ray-set terminates, the last intersection tree is discarded and the tree that was just generated in turn is retained, to play the role of intersection-guide for the next ray-set.

However, we must be able to detect cases such as those illustrated in Figure 2, as we mentioned; this can be done as follows. A logical "container", a kind of "safety zone", can be constructed around every ray in a ray-set. These containers will be centred around the ray with which they are associated and extend outward *to the nearest object not intersected by that ray* (Figure 4). It can be seen that if a corresponding ray from the next ray-set does not "pierce" (intersect) the side of the relevant container, and intersects the same object intersected by this ray, then that object must be the foremost object intersected by the corresponding ray. Thus, if each ray in the last ray-set has associated container information, the situation shown in Figure 2 can be avoided: rays in the next ray-set that pierce the container of their corresponding ray or fail to intersect the object intersected by that ray, require a region partition, as in standard ray-tracing. Rays that do not pierce the relevant container, on the other hand, and intersect the object intersected by their corresponding ray do not require a region partition.

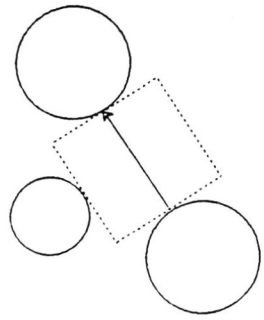

Figure 4 - "Safety Zone" Around a Ray (Cross-sectional View)

Since the savings that can be achieved by this strategy depend on trading the cost of checking every object in the region for the cost of "pierce-checking" the container, it is important that the latter operation be as computationally simple as possible. A radially-symmetric container is a great help in achieving this goal. For this reason, we chose *cylinders* for the containers. Such a cylinder starts at the point of origin of a ray, has its central axis aligned with it, and ends at its point of intersection with an object.

2.3.2. Container/Cylinder Construction

It is not difficult to construct a cylinder like the one just described. Notice that every bounding volume in a region becomes a member of one and only one of the sets B, N and I, defined in Section 2.1, in the course of a region partition for some ray. Let us now consider the set N. If it is not empty, then we can simply check each element in the set to find any that is between the bounding plane of the ray and a parallel plane that passes though the nearest object intersection point. Of these, we take the distance of the volume that is nearest to the ray, radially, as the cylinder radius (Figure 5). The cylinder has its central axis aligned with the ray, which we call the *formative ray* for the cylinder, and is bounded by the two planes mentioned. We store the cylinder information in an intersection tree node simply by storing the radius defined above and the ray direction. The other attributes (the cylinder bounds) are defined implicitly by information already stored in the intersection tree by the standard algorithm,[25] namely the originating object and the nearest object intersected by the formative ray.

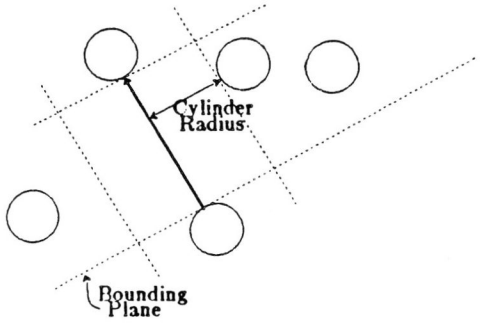

Figure 5 - Safety-Cylinder Construction

2.3.3. Containers/Cylinders for Light Sources

To compute shadows, rays are traced from object intersection points in the direction of each light

source, as mentioned in Section 2.2. If one of these rays intersects an object "en route", the point in question is in shadow with respect to that source. Rays traced for this purpose are somewhat different than those traced for object intersection in that no rays are traced recursively from any object that they intersect.

Coherence cylinders can also be constructed and used for these light-testing rays. If an object is intersected by such a ray, a cylinder is built in the manner discussed earlier (Figure 5).

2.4. Summary

In summary, our algorithm differs from the standard one in two major ways: First, the standard intersection tree is retained from pixel to pixel as the program runs. The objects intersected by rays of previous ray-sets are used to suggest which objects might be intersected by corresponding rays in the current ray-set. Second, logical cylinders are constructed using information obtained during a region partition. These are used to indicate when the intersection information from a previous formative ray is no longer useful and that a region partition will be required.

3. Probabilistic Analysis

We now consider two major questions: first, for a scene with a given percentage of coherent rays, what order of computational savings can be achieved using the new algorithm and, second, since it has additional costs beyond standard ray-tracing (due to cylinder construction and pierce-checking), what percentage of rays must be coherent for the new algorithm to outperform standard ray-tracing?

Beginning with the first question, a simple argument can be made to derive an upper bound on the savings that can be achieved using the new algorithm. Suppose that a ray found to be coherent cost *nothing* computationally. Denote the time required to render a given scene using standard ray-tracing as T_{ST}. Then, if a fraction C of the number of rays traced by the algorithm are coherent, an expression for the amount of time saved is $Savings = C * T_{ST}$. This is an upper bound since any implemented algorithm must do some work to process even a coherent ray.

From the above, we note that *the savings that can be achieved by the new algorithm are linear in the amount of coherence present.* For example, if half the rays traced in a scene are found to be coherent ($C = 0.5$), computation time can be reduced by no more than 50%.

We now turn to the second question, concerning the percentage of coherent rays needed before the new algorithm shows a savings. We start by deriving expressions for the amount of time needed to process a ray in the standard algorithm and in the new one.

In standard ray-tracing, there are two costs associated with each ray. First, a scene partition must be performed, at a cost proportional to the number of volumes n in the scene. Second, the volumes that are intersected must be checked to find the one that is nearest to the ray anchor. The time needed to determine this is a weak function of n due to the fact that on average, few volumes will need to be checked. Therefore, we treat this time as a constant, α. We also say that this constant includes the cost of computing the directions of any recursive rays. Altogether then, the amount of time needed for each ray in standard ray-tracing, which we denote t_{ST}, is $t_{ST} = kn + \alpha$.

We now look at the cost of a ray in the new algorithm. There are two cases to consider, corresponding to whether or not a safety cylinder must be constructed. The cost of a ray when a cylinder must be constructed is similar to the cost of a ray in standard ray-tracing: the scene must be partitioned and the nearest intersected object found. An amount of additional work λn must then be done to determine the cylinder radius. Thus, the time needed to process a ray in this case is $t_{cylinder} = kn + \alpha + \lambda n$. Simplifying in terms of t_{ST} yields $t_{cylinder} = t_{ST} + \lambda n$.

The cost of a coherent ray, on the other hand, is simply the cost of computing the intersection point with the object intersected by the formative ray, together with the cost of computing any recursive rays. This cost is less than or equal to α. (The cost of detecting that a cylinder is pierced is negligible compared to the cost of a scene partition, for any scene containing more than a few objects.)

We can now give an expression for the maximum cost of a ray in the new algorithm, t_{CT}. Denoting the probability of a ray being coherent as C, the time needed for its processing is

$$t_{CT} \leq C(\alpha) + (1-C)t_{cylinder}, \text{ for } 0 \leq C \leq 1.$$

Rearranging yields

$$t_{CT} \leq t_{cylinder} - C(t_{cylinder} - \alpha), \text{ for } 0 \leq C \leq 1.$$

Now, by setting t_{ST} equal to t_{CT} and solving for C, we can find the threshold value C_T at which the new algorithm costs less than standard ray-tracing.

$$t_{ST} \leq t_{CT}$$
$$\leq t_{cylinder} - C(t_{cylinder} - \alpha)$$

Solving for C yields

$$C_T \leq \frac{t_{cylinder} - t_{ST}}{t_{cylinder} - \alpha}$$

By substituting in for $t_{cylinder}$ and t_{ST}, this reduces to

$$C_T \leq \frac{1}{1 + \frac{k}{\lambda}}$$

If the coherent fraction of all rays traced is greater than the maximum value of C_T, the new algorithm will outperform standard ray-tracing. To examine this further, we look again at the last equation. It is clear that the larger the ratio of k/λ, the lower the threshold value will be. We will discuss this ratio further in the next section.

Finally, in the worst case, the new algorithm will require more computation time than the standard algorithm. Consider a scene in which rays are *never* coherent, for example. In such a case, partitions are required for every ray, as in the standard algorithm but, in addition, work must be done to construct cylinders and check for piercing. Zero coherence is clearly the limit case as the number of objects in the scene increases. This underscores the point that coherent ray-tracing can only be considered for low- or moderate-density scenes.

4. Implementation Results

We have implemented the new algorithm in Pascal under Berkeley Unix 4.2 BSD. The computers used were Digital Equipment Corporation's VAX-11/750 and VAX-11/780.

We rendered a group of scenes using two versions of the new algorithm. The first version was the new algorithm as presented above. The second version was a modification of the new algorithm to simulate standard ray-tracing. The modification was accomplished by commenting out all code dealing with coherency; the N set was not retained or processed and the intersection tree for a given image point was discarded after the overall colour computation for that point.

Both versions incorporate the same recursion-halting criteria. This includes the standard criteria such as "nothing intersected" or "intersection with an opaque, non-transparent object", as well as the "dynamic recursion termination" condition mentioned in Section 2.2.[8,11]

The programs were tested on four scenes containing an increasing number of randomly-positioned spheres. The results are tabulated below. It can be seen that the coherent technique is no better than the standard algorithm for scenes containing on the order of 8 or 9 spheres.

Results				
Number of spheres	CPU time, std. alg (hrs)	CPU time, coher. alg (hrs)	Percent rays coherent	Ratio, coherent / std.
1	0.283	0.232	76.0%	.80
2	0.552	0.48	74.7%	.86
4	1.351	1.208	78.0%	.875
8	3.396	3.539	66.2%	1.02

Table 1.

Perhaps the most important result in this paper is contained in the last row of Table 1. It shows that *despite the fact that nearly two-thirds of the rays behaved coherently, the new algorithm produced no savings over standard ray-tracing.* How can this be the case? There are two likely answers.

First, pierce-checking even as simple a shape as a cylinder is not cheap; the cost is about as much as two ray-sphere intersection tests. Thus, even in the absence of other negative factors, there would have to be a certain number of objects in a scene before a savings over standard ray-tracing could be realized.

However, as the number of objects in a scene increases, the average cylinder radius and length *decreases*; more and more time is spent constructing and checking cylinders that will be only be pierced. This means that ultimately, for some "crossover" number of objects, coherent ray-tracing will inevitably cost more than the standard method. The only question is whether that crossover value is large enough that the technique is of practical value. Table 1 shows that this is not the case.

Combining the last line of Table 1 with the final equation of Section 3 shows that the ratio k/α is approximately 0.5. Recall that k is the proportionality constant for a scene partition while α is a similar constant for cylinder construction. Notice that if α could be reduced relative to k, C_T could also be reduced; this in turn implies that coherent ray-tracing could be applied to scenes with less coherence than the amount in the 8-sphere scene. However, since cylinder construction is already done by simple comparisons, it is difficult to see how the speed of this operation could be significantly increased.

5. Conclusions

An algorithm for coherent ray-tracing has been presented. The algorithm uses coherence to reduce the average amount of computation required to construct the intersection tree needed for colour computation. An analysis and empirical study of the algorithm was performed. The results show that the algorithm fails to out-perform standard ray-tracing on scenes of practical size. This indicates that other algorithmic techniques must be considered in order to reduce the large computational cost of ray-tracing.

Acknowledgments

The authors would like to thank Mark Dippé and the other members of the Berkeley Computer Graphics Lab for fruitful discussions.

References

1. Arthur Appel, "Some Techniques for Shading Machine Renderings of Solids," pp. 37-45 in *Proceedings of the Spring Joint Computer Conference, Vol. 32*, AFIPS, Thompson Books,Washington, D.C.(1968).
2. Peter R. Atherton, "A Scanline Hidden-Surface Removal Procedure for Constructive Solid Geometry," pp. 73-82 in *SIGGRAPH '83 Conference Proceedings*, ACM,(July, 1983).
3. Chris Brown, "Special Purpose Computer Hardware for Mechanical Design Systems," pp. 403-414 in *Proceedings of the 1981 National Computer Graphics Association Conference*, National Computer Graphics Association, Inc.,Washington, DC.
4. Arthur G. Chang, *Parallel Architectural Support for Raytracing Graphics Techniques*, Master's Thesis, Computer Science Division, EECS Department, University of California, Berkeley, Berkeley, California.
5. John G. Cleary, Brian Wyvill, Graham M. Birtwistle, and Reddy Vatti, *Multiprocessor Ray Tracing*, Technical Report No. 83/128/17, Department of Computer Science, The University of Calgary (October, 1983).
6. Hiroshi Deguchi, Hitoshi Nishimura, Hiroshi Yoshimura, Toru Kawata, Isao Shirakawa, and Koichi Omura, "A Parallel Processing Scheme for Three-Dimensional Image Creation," in *Proceedings of the International Symposium on Circuits and Systems*, IEEE,Montreal(1984).
7. Mark E. Dippé and John A. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," pp. 149-158 in *SIGGRAPH '84 Conference Proceedings*, ACM,Minneapolis(July 23-27, 1984).
8. Patrick A. Fitzhorn, *Realistic Image Synthesis: A Time Complexity Analysis of Ray Tracing*, Master's Thesis, Colorado State University, Fort Collins, Colorado (Spring, 1982).
9. Andrew S. Glassner, "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications*, Vol. 4, No. 10, October, 1984, pp. 15-22.
10. Robert Goldstein and Roger Nagel, "3-D Visual Simulation," *Simulation*, Vol. 16, No. 1, 1971, pp. 25-31.
11. Roy A. Hall and Donald P. Greenberg, "A Testbed for Realistic Image Synthesis," *IEEE Computer Graphics and Applications*, Vol. 3, No. 8, November, 1983, pp. 10-19.
12. Patrick M. Hanrahan, "Raytracing Algebraic Surfaces," pp. 83-90 in *SIGGRAPH '83 Conference Proceedings*, (July, 1983).
13. Paul Heckbert and Pat Hanrahan, "Beam Tracing Polygonal Objects," pp. 119-129 in *SIGGRAPH '84 Conference Proceedings*, ACM,(July, 1984).
14. James T. Kajiya, "New Techniques for Raytracing Procedurally Defined Objects," *ACM Transactions on Graphics*, Vol. 2, No. 3, July, 1983, pp. 161-181.
15. James T. Kajiya, "Ray Tracing Parametric Patches," pp. 245-254 in *SIGGRAPH '82 Conference Proceedings*, (July, 1982).
16. Douglas S. Kay, *Transparency, Refraction, and Ray Tracing for Computer Synthesized Images*, Master's Thesis, Cornell University, Ithaca, N.Y. (January, 1979).
17. Yong Tsui Lee and Aristides A. G. Requicha, "Algorithms for Computing the Volume and Other Integral Properties of Solid Objects, I : Known Methods and Open Issues, and II: A Family of Algorithms Based on Representation Conversion and Cellular Approximation," *Communications of the ACM*, Vol. 25, No. 9, September, 1982, pp. 635-650.
18. Nelson L. Max, "Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset," pp. 317-324 in *SIGGRAPH '81 Conference Proceedings*, (August, 1981).
19. Scott D. Roth, "Ray Casting as a Method for Solid Modelling," *Computer Vision, Graphics and Image Processing*, Vol. 18, No. 2, February, 1982, pp. 109-144.

20. Steven M. Rubin and J. Turner Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," pp. 110-116 in *SIGGRAPH '80 Conference Proceedings*, ACM,(July, 1980).

21. Kim L. Shelley and Donald P. Greenberg, "Path Specification and Path Coherence," pp. 157-166 in *SIGGRAPH '82 Conference Proceedings*, (July, 1982).

22. Ivan E. Sutherland, Robert F. Sproull, and Robert A.. Schumacker, "A Characterization of Ten Hidden Surface Algorithms," *ACM Computing Surveys*, Vol. 6, No. 1, March, 1974, pp. 1-55.

23. Michael Ullner, *Parallel Machines for Computer Graphics*, Ph.D. Thesis, California Institute of Technology, Pasadena, California (1983).

24. J. Turner Whitted, "Processing Requirements for Hidden Surface Elimination and Realistic Shading," pp. 245-250 in *IEEE Compcon Digest of Papers*, (Spring, 1982).

25. J. Turner Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, Vol. 23, No. 6, June, 1980, pp. 343-349.