FACTORS AFFECTING PERFORMANCE OF RAY TRACING HIERARCHIES

K. R. Subramanian and Donald S. Fussell

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

TR-90-21 July 1990

Factors Affecting Performance of Ray Tracing Hierarchies

K. R. Subramanian   Donald S.Fussell

Department of Computer Sciences
The University of Texas at Austin
Austin, Tx 78712

Abstract

In this article, we study some of the important characteristics that affect the performance of ray-tracing hierarchies. Location and orientation of space partitioning planes, the role of bounding volumes, void ares in the nodes of the hierarchy and hierarchy traversal methods are studied individually to determine their effect on performance. We then demonstrate how these characteristics can be combined to obtain object hierarchies that can out-perform some of the best known ray tracing hierarchies. Results are shown using standard ray-tracing benchmarks.

CR categories and Subject Descriptors:
I.3.3 [Computer Graphics]: Picture/Image Generation;
I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism;

General Terms: Algorithms, Graphics.

Additional Key Words and Phrases: Ray tracing, computer graphics, bounding volume, extent, partitioning plane, hierarchy, search structure, traversal, characteristics.

1 Introduction

In recent years, ray tracing has established itself as an important rendering technique. Its ability to determine visible surfaces, compute shadows, model reflection and transmission of light and a host of other effects [5] has contributed to its growing use. However, ray tracing, if not carefully done, can be a computationally expensive technique. Consequently a great deal of research has focused on discovering efficient ways to perform ray tracing.

The principal expense in ray tracing lies in determining a ray's first (i.e., closest to the ray's origin) intersection with an object in a scene. This must be done several times per pixel, the exact number depending on the effects being generated and the scene being rendered. Research on efficient algorithms has quite properly focused on minimizing the cost of these intersection calculations. Bounding volumes [16][19][24][25], space partitioning structures [8][10][15], item buffers [24], shadow buffers [13] and ray coherence techniques [1][14][20] all have proven effective at improving the efficiency of ray tracing.

Many of these techniques have employed data structures for speeding up the search for a closest intersection on a ray. Data structures which support efficient geometric search allow us to look at only a small percentage of the scene to determine the closest intersection. Octrees [10], BSP trees [15], and nested bounding volumes [11] are examples of explicitly hierarchical search structures of this type, while the uniform subdvision techniques [8][4] use non-hierarchical search structures.

While all these search structures are being used with great success, performance comparisons between them until now have been solely via timing benchmarks. Comparisons between different methods have been difficult, since each method was tested by researchers on their own set of test cases, thus precluding any meaningful results. A first step has been taken in this regard by Haines [12], in proposing a set of test scenes with detailed specifications of the the various parameters of any ray trace. If two methods use the same test case with the exact same parameter values, the run times obtained from the ray trace are a better measure of their respective performances.

In this paper, we bring out some of the important characteristics that affect the performance of search structures used in ray tracing. Locations of partitioning planes, partitioning dimension, effects of bounding volumes, presence of void areas (void areas refer to sections of the environment where there are no object primitives) and hierarchy traversal methods are some of the characteristics examined. The most commonly used search structures are then described in the context of these characteristics. This will give us a better insight into the characteristics exploited (or not exploited) by each search structure and help us relate performance to scene characteristics. Finally, it allows us to explore the possibility of combining different characteristics to improve the performance of existing schemes and design better object hierarchies. We finish by justifying our results with carefully designed experiments.

2 Search Structures Used in Ray Tracing

Some of the common hierarchical structures being used in ray tracing include BSP trees [15], octrees [10], and the surface area based heuristic described in [17]. In addition to these, we consider a hierarchical technique related to the first two which uses *k-d* trees [9][21] We begin with a brief description of each of these methods.

A BSP tree is any binary tree structure used to recursively partition space. BSP trees have been used extensively to determine visible surfaces [22][6][7]. In Kaplan's [15] implementation of the BSP tree for ray tracing, axis-aligned planes are used to partition space. At each step of the subdivision, three slicing planes are used to divide space into eight equal sized octants (or voxels, which are just rectangular parallelepipeds). Each object in the scene is tested against each octant to see if any part of its surface intersects it. If it does, then it is added to the list of objects for that node. Voxels that contain large numbers of objects are recursively subdivided until each subdivided region has less than a small number of objects or its size becomes smaller than a set threshold. The subdivision is adaptive, in that only voxels that contain primitives are subdivided. Thus, the structure adapts itself to the input scene.

To determine a ray-object intersection from the Kaplan-BSP tree, the origin of the ray is compared against the partitioning planes in the tree, starting from the root of the tree. The leaf node (or voxel) containing this point can be determined. All primitives in this voxel are intersected with the ray. If an intersection is found within this voxel, the closest of these is the required intersection. Otherwise, the face through which the ray exits this voxel is determined by intersecting the ray against all the six faces of this voxel. The ray is then extended a small amount, depending on the size of the smallest voxel in the tree. This end point is put back into the root of the tree to determine the next voxel visited by the ray and the search continues.

The octree hierarchy used by Glassner [10] performs a subdivision identical to Kaplan's BSP tree. Each level of the octree corresponds to three levels of the BSP tree. The difference lies in the way Glassner stores the octree. While Kaplan builds a binary tree, Glassner uses a hash table with link lists, which results in considerable savings in pointer space. The traversal is also identical to the BSP tree.

The *k-d* tree [2][3] hierarchy used in [9] is another special case of the BSP tree. It also uses axis-aligned partitioning planes, but it has greater flexibility in their location as well as the choice of the partitioning dimension. The initial node is simply the scene extents. Nodes are partitioned by planes whose locations are chosen by performing a binary search of potential plane positions within the node extents along all three dimensions. The plane that best balances the number of primitives on both sides of the partition and contains the least number of primitives straddling it is the desired partitioning plane. Subdivision continues as long as there is at least one primitive on each side of the partitioning plane and at least one of them is completely on one side of the plane. Insertion of a plane creates a pair of nodes, the union of whose extents equals the parent's extents. In cases where a new node's extents are much larger than the bounding box of the objects included in the node, this bounding box is stored in the node.

To traverse the *k-d* tree, a ray is intersected with the partitioning plane stored at the root of the hierarchy. By looking at the ray parameters (its origin and direction), it is simple to determine the order in which the regions should be examined. Regions are examined in order of increasing distance from the ray origin. If the ray intersects the bounding volume for a node (either its extents as determined by partitioning planes or the bounding box of its included objects, as appropriate), the node's children are processed next, otherwise the node's siblings which are further from the ray origin are processed next. The entire traversal can be done in ray parameter space. This traversal can also be used with the octree or BSP tree hierarchies.

3 Characteristics of Ray Tracing Hierarchies

Hierarchical structures help speed up the search for a ray's first intersection with an object in the scene. Two important techniques that are used to achieve this are space partitioning and bounding volumes.

3.1 Space Partitioning

In space partitioning structures, object space is divided up into a collection of convex regions. Axis-aligned planes are the most common means used [15][10], although planes of other orientations have been used to solve the hidden surface problem [7][22]. The search for the closest intersection usually involves intersecting the partitioning planes. Axis-aligned planes are advantageous since they are less expensive to intersect when compared to planes that are arbitrarily oriented. The process is recursive, i.e., the regions are recursively partitioned until they violate the termination criteria used. An example is shown in in Fig. 1, with two levels of partitioning.

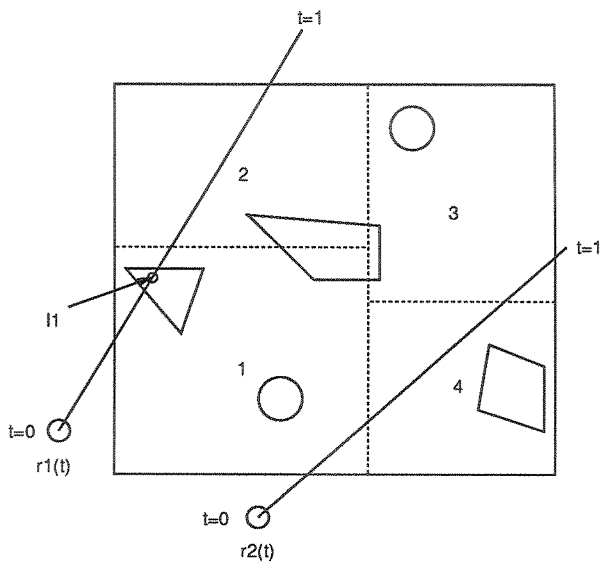

Figure 1: Sample Rays

There are important advantages to using a space-partitioning structure.

1. It enables the use of an ordered search for the closest ray-object intersection. The preferred order is along the path of the ray, so that the search can be terminated as soon as an intersection is found.

2. The partitioning planes help identify the regions visited by each ray. This is a small fraction of the total number of regions in the search structure. Thus large sections of the scene remote from the ray are never examined. In Fig. 1, ray $r_1(t)$ examines region 1, while $r_2(t)$ examines regions 1, 4 and 3. With increased partitioning, the number of regions actually examined will be a small fraction of the total.

3. The structure is direction independent. With the help of the partitioning planes, an ordering can be determined for a ray with arbitrary origin and direction. Thus all kinds of spawned rays like shadow rays and reflection rays can be traced with no change in the search structure. In Fig. 1, ray $r_1(t)$ finds an intersection I_1 and terminates, while

$r_2(t)$ examines regions 1, 4 and 3 and terminates without finding an intersection.

4. The partitioning is adaptive, i.e. only parts of the scene that contain large collections of primitives are subdivided. This is important when the object distribution is not uniform. If space were subdivided uniformly, then then there would be large collections of empty voxels, and traversal costs would become prohibitively high.

3.2 Location and Orientation of Partitioning Planes

At each stage of space partitioning, we need to decide the location and orientation of the plane. The most common hierarchies like the octree and the Kaplan-BSP tree use axis-aligned planes which bisect the node extents in each dimension at all nodes of the hierarchy. In the octree all three dimensions are partitioned simultaneously while in the BSP structure the partitioning dimension cycles through the X,Y and Z axes. The *k-d* tree method [9] uses a median-cut scheme to locate the plane. The planes are still axis-aligned but they are located so as to place equal numbers of object primitives on both sides of the planes. At the same time, the number of primitives crossing the plane is minimized. Another heuristic used in [17] takes into account the surface area at each node of the hierarchy. Rather than balancing the number of objects on either side of the partitioning plane, the sum of the product of the surface area and the object count on both sides of the plane is minimized.

Each of these approaches has its own strengths and weaknesses. Consider the BSP or octree subdivision. Since the planes are fixed to be at the centers of the node extents, the partitioning might create a large number of empty voxels, especially when the scene distribution is far from uniform. This means traversing empty regions, an expense to be avoided, since no intersection can be found in these regions. Also, the inflexibility in plane locations might cause the plane to cross a large number of primitives, all of which must be counted on both sides of the hierarchy (splitting up primitives presents no advantage). However, the

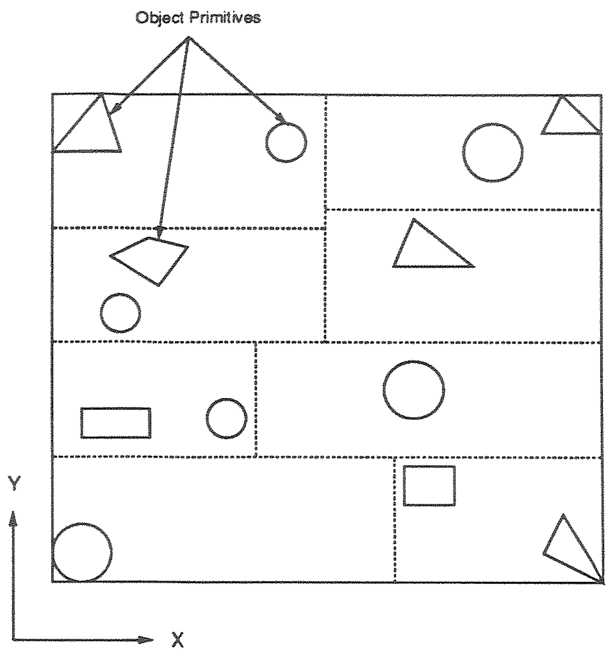

Figure 2: K-d Tree subdivision

partitioning is adaptive in two respects; a region is not partitioned if it contains less than a small number of objects, thus avoiding the creation of large numbers of regions containing few or no object primitives, and regions smaller than a preset size are not subdivided any further. The size of the region determines the number of rays that will intersect it. The smaller it is, the fewer the number of rays that will intersect the region. To sum things up, the creation of empty voxels is itself a source of inefficiency in the first place since no intersection can be found in such voxels, but the fact that they are not subdivided any further is a desirable factor since it helps the structure adapt itself to the locations and densities of the primitives in the input scene.

In the *k-d* tree hierarchy, the planes are located so as to balance the number of primitives on either side of the plane (Fig. 2). A binary search is conducted to determine the plane that best balances the primitives and, at the same time, minimize primitives that straddle the

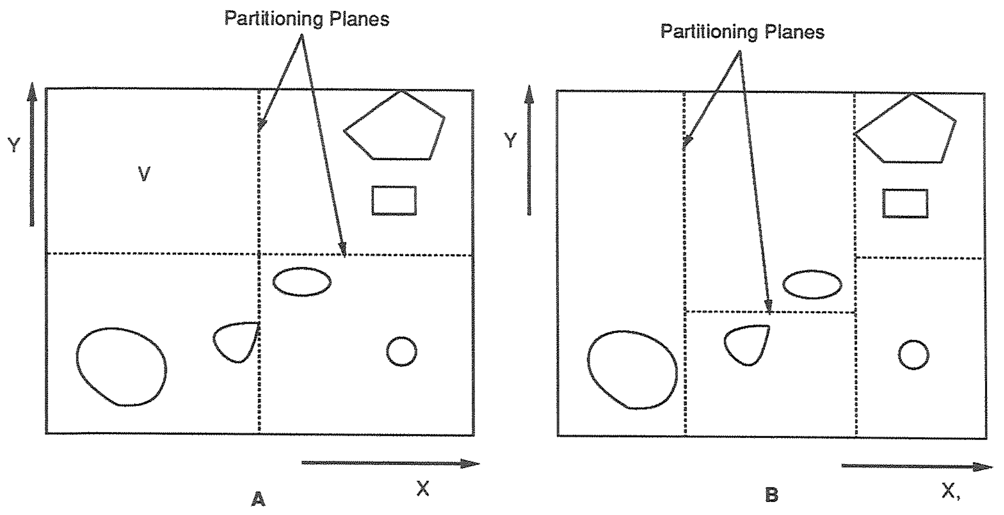

Figure 3: Kaplan-BSP vs. median-cut subdivision

plane. The partitioning dimension is another parameter that makes the plane choice even more flexible. This automatically makes the partitioning adaptive since the plane is going to be located in the vicinity of the object primitives. The median-cut approach results in a hierarchy that has a smaller average height than the Kaplan-BSP scheme since it builds a balanced binary tree. This is desirable since getting to the leaf nodes (and hence, object primitives) is faster and less expensive. The main disadvantage of this scheme (as presented so far) lies in its potential to create large void areas in its regions, especially when the object distribution is not uniform. In this case, the *k-d* tree approach will have larger void space and, in general, poorer performance.

Consider the simple scene in Fig. 3. In 3a, Kaplan's BSP subdivision with two planes creates the void space marked V. This space is not subdivided any further since it does not contain any object primitives. However in 3b, the median-cut approach always tries to balance the primitives. Thus the void space that appeared in the BSP subdivision will never be isolated by the median-cut scheme because planes are always located in the vicinity of the objects. While this is advantageous in cutting down the cost of reaching the leaf nodes during hierarchy traversal, it allows large collections of the rays to intersect with these void spaces, thus increasing the traversal cost.

We will later show how this problem can be overcome with the use of bounding volumes.

The heuristic proposed by Macdonald and Booth [17] considers two different characteristics, surface areas of the nodes in the hierarchy and their object count. Let b be a partitioning plane of a hierarchy. $SA_l(b)$ and $SA_r(b)$ represent the surface areas on the left and right sides of the hierarchy. n_l and n_r are the object counts on the two sides of the hierarchy. The function to minimize is given by

$$f(b) = SA_l(b).n_l + SA_r(b).n_r \tag{1}$$

A good feature of this heuristic is that it takes into account the size of objects. The idea is to strike a balance between surface area and the number of objects that make up that area on each side of the plane. However, this heuristic also fails to address the problem of void area. Once a partitioning plane is located at a node, both sides of the plane can potentially contain large void areas. These areas must be pruned from the hierarchy. One way to do this is to use bounding volumes at each node of the hierarchy. Now the surface area of a node is redefined to be the surface area of the bounding volume of the subtree of objects stored at that node. An important factor to note is that $(SA_l(b) + SA_r(b))$ will, in general, be less than the surface area of the bounding volume of the original node. The difference is a measure of the void area at the node. We will investigate the effects of each of these characteristics in our experiments.

3.3 Traversal Methods

Earlier, we described two different methods to traverse the BSP tree, one used by the Kaplan-BSP tree and Glassner's octree, and the one used by the k-d tree. The regions

Operation	flops
Reaching leaf node	h_{avg}
Determining face through which ray exits	15
Computing exit point	6
Extending ray into the next region	9
Total cost/region examined	$30 + h_{avg}$

Table 1: BSP/octree Traversal Operations

Operation	flops
Plane intersection at each node	2
Branching decision at each node	1.5
Total cost to get to leaf node	$3.5h_{avg}$

Table 2: $K - d$ Traversal Operations

identified by both methods for each ray trace are identical. The difference lies in how this is accomplished. Let us look more closely at the operations performed by the two methods (the numbers given in Tables 1 and 2 pertain to our implementation and can be replaced by equivalent quantities for other implementations).

Tables 1 and 2 show the operations performed in both of these traversal methods. h_{avg} is the average height of the hierarchy. For the first method, the total operations is an average count while in the second method, the total operations is a worst case count. Most often, neighboring regions will be close to each other in the hierarchy. Thus, if the next region to be processed is close to the region just processed in the hierarchy, then both regions will have a common ancestor that is relatively deep in the hierarchy. Hence the number of plane intersections and branching decisions done to get to the new region will be much smaller than h_{avg}.

Looking at the expressions for the two methods, it is not clear which method is better. The octree/BSP traversal does more work in moving from voxel to voxel while the *k-d*

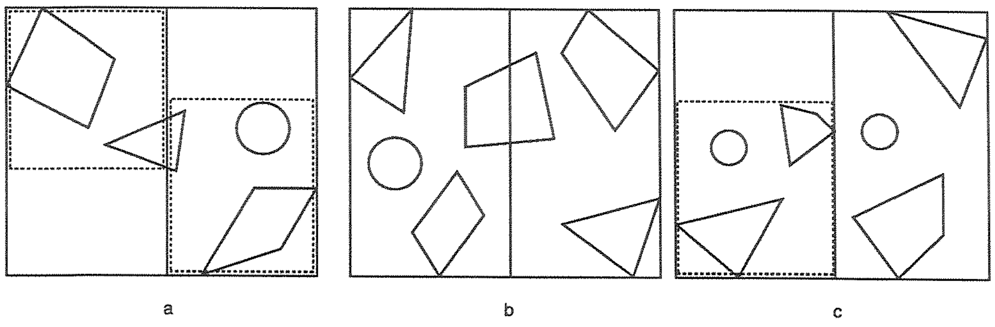

Figure 4: Bounding Volumes

traversal is doing more work getting down to the leaf nodes. For very dense scenes, if an intersection is found within the first few regions the ray encounters, then the first method does less work. This is because the *k-d* tree method is determining the order (in advance) for regions that may never be visited by the ray because of early termination. If the ray travels a number of regions and if there is locality properties between successive regions in the hierarchy, then the second method will do better because all its computation is done in ray parametric space and no coordinates are computed. One other restriction for the BSP/octree traversal is that all the regions visited must be contiguous, since the ray is 'extended' from region to region. The amount by which the ray is extended is determined by the smallest voxel in the hierarchy. If there were gaps between voxels, this traversal could put a ray origin in a gap and there would be no corresponding voxel containing the origin. The *k-d* tree traversal has no such restriction. This becomes important when we start using bounding volumes in hierarchies.

3.4 Role of Bounding Volumes

There are some important advantages to using bounding volumes to optimize ray tracing.

1. Around object primitives, bounding volumes provide a simple and inexpensive non-intersection test. Only if the ray intersects the bounding volume does the primitive

inside it needs to be tested for intersection.

2. By surrounding clusters of objects with bounding volumes, large sections of object space can be pruned away, drastically reducing the ray search space. Bounding volumes are more effective in this respect, since, in general, they provide tighter enclosures around object clusters than those created by space partitioning structures. Also, the bounding volume can be made arbitrarily tight [16], although the cost of intersecting the bounding volume also increases proportionately. Grouping objects into clusters of increasing size and surrounding them with bounding volumes results in a bounding volume hierarchy. Any node in this hierarchy represents a cluster of objects contained in its subtree.

3. Bounding volumes can provide a much simpler representation for objects that are located far from the eye [23]. Thus complex geometric calculations can be avoided if the object is going to occupy only a small fraction of a pixel in the image plane.

4. The algorithms used to construct bounding volumes are critical to performance. The best known hierarchy to date is the Automatic Bounding Volume hierarchy [11]. Its biggest advantage is the automatic construction of the hierarchy, which facilitates rendering complex environments.

3.5 Discussion

Space partitioning hierarchies optimize ray tracing by providing an ordering of regions visited by the ray independent of the ray origin or direction. This ordering also makes it easy to terminate a ray trace once the closest intersection is determined. Since only regions visited by the ray are examined for intersection, the structure helps avoid examining regions that are far from the path of the ray. Space partitioning structures are adaptive, concentrating the partitioning in the vicinity of objects. Since all the partitioning planes are axis-aligned, they have the potential to create large void spaces which are a source of inefficiency. Bounding volume hierarchies, on the other hand, optimize ray tracing by culling away large sections of object space and provide a compact representation for the objects at every node of the hierarchy. They are more effective than space partitioning hierarchies in this because of the greater freedom available in choosing the location, orientation and shape of the bounding volume. Also, they can be made to fit as tight as possible around object clusters thus reducing the ray search space even further. Rays that do not intersect a bounding volume in the hierarchy will not need its subtree (containing bounding volumes and/or object primitives) to be examined any further. Lastly, the bounding volume hierarchy traversal is not along the path of the ray, although a partial ordering can be obtained by sorting and processing bounding volume intersections in sorted order [16], unlike space partitioning hierarchies.

Since the optimizations provided by space partitioning hierarchies and bounding volumes seem to be orthogonal to each other, can we combine them to good advantage? Hierarchy traversal methods described earlier also need be examined in more detail to determine how they perform with typical scene models. Lastly, location of partitioning planes is also a characteristic that merits a closer look. Our goal will be to attempt to combine these optimizations to build a space partitioning hierarchy that will be capable of outperforming any hierarchy that uses only some of these characteristics.

4 Implementation

We have implemented a space partitioning hierarchical structure to investigate the different characteristics that affect performance. All experiments were conducted on a Sun 4 (SPARC) workstation running 4.0.3 UNIX[1]. All the images were computed at 512 by 512 resolution. Images are illustrated in the color plates at the end of the paper. Run times are in minutes of cpu time.

[1]UNIX is a trademark of AT&T Bell Laboratories.

Partitioning	Bounding Volumes	Traversal
BSP	No	*k-d*
BSP	Yes	*k-d*
BSP	No	BSP/octree
BSP	No	*k-d*
k-d (median-cut)	No	*k-d*
k-d (median-cut)	Yes	*k-d*
SA heuristic	No	*k-d*
SA heuristic	Yes	*k-d*

Table 3: Experiments Conducted

Ray counts are in thousands

Scene	Tetra	DNA	Arches	Spd.Balls	Spd.Tree	Spd.Mount
Objects	1024(P)	410(S)	4818(P)	7382(PS)	8191PSC	8196(PS)
Lights	1	1	2	3	7	1
Total rays	299	445	437	1819	1676	1393
Visual rays	262	323	317	722	452	977
Shadow rays	37	122	120	1097	1224	416
Hit rays	44	75	73	873	242	683

Table 4: Test Scenes

Table 3 describes the tests we have conducted to bring out the performance characteristics that we described in the previous section. In Table 3, 'SA heuristic' refers to the surface area heuristic used in [17], described by Equation 1. BSP/octree traversal refers to the traversal methods used by Glassner [10] and Kaplan [15]. *k-d* traversal was used in [9][18], as we described earlier. Seven different datasets were used as test cases. Most of these are scenes proposed as benchmarks [12]. Details of these datasets and ray statistics are described in Table 4.

In our implementation, the partitioning plane can be located so as to bisect the scene extents (BSP/octree), balance the number of objects on either side (*k-d*), or search for the best plane between the spatial and object medians (Surface area heuristic). Bounding volumes can be turned on or off for any of these schemes. The *k-d* traversal can be used with any of these schemes, while the BSP/octree traversal can be used only in the absence of bounding volumes in the hierarchy. This is because the BSP/octree traversal requires that voxels be contiguous, which is no longer the case once bounding volumes are introduced into the hierarchy.

Bounding volumes used in the hierarchy are axis-aligned parallelepipeds. Object primitives are also surrounded by parallelepipeds. Bounding volumes are used in the hierarchy only when their presence at a node will result in a significant reduction in void space. Some 2D examples are shown in Fig. 4. In 4a, bounding volumes are required on both sides of the plane, in 4c, no bounding volumes are required since no reduction in void space is achieved by doing so, and in 4c, only the left side needs it.

We measure the floating point operations spent in computing bounding volume intersections (both in the hierarchy and those surrounding object primitives), traversal operations (for example, plane intersections) and object intersections. This constitutes the majority of the work spent in ray tracing. This is a measure of performance independent of the computer used to run the ray trace. We also compute the total void area for each scheme. This helps bring out the role of bounding volumes in the hierarchy. To compute this, the area of the node is first determined, which is just the surface area of its bounding volume. Once a partitioning plane subdivides this node, we have two sub-nodes. The tightest bounding volume enclosing all the primitives in each sub-node is computed, clipped to the bounding volume of the original node and the partitioning plane. The void area is the difference between the area of the original node and the sum of the areas of its two children. This computation is carried out at each node that is subdivided to get the total void area.

5 Experimental results

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	11.45	43388.03	153.20	2.90
DNA	11.70	32552.10	240.86	6.67
Arches	14.41	59.12	428.22	9.60
Spd.Balls	20.07	3034.28	547.93	69.93
Spd.Tree	24.52	48611.76	484.42	73.22
Spd.Mount	19.53	344.31	273.61	31.58

Table 5: BSP subdivision, no bounding volumes, *k-d* traversal.

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	8.78	105095.42	200.33	3.27
DNA	11.14	60220.91	268.71	7.21
Arches	13.56	252.38	894.52	20.35
Spd.Balls	14.97	43655.13	708.05	84.39
Spd.Tree	14.02	716925.55	1097.19	108.88
Spd.Mount	13.97	1617.71	600.70	43.38

Table 6: *k-d* subdivision, no bounding volumes, *k-d* traversal.

Tables 5, 6 and 7 illustrate the results for the different space partitioning schemes in the absence of bounding volumes. Among these, the BSP (or octree) scheme performs best. The reason for this is the larger void areas in the *k-d* and SA methods (also illustrated in the tables). Performance is tracked reasonably well by the void area figures.

To counter the problem of void area, we next introduce bounding volumes into the hierarchy. The results are illustrated in Tables 8, 9 and 10. The performance of all the schemes improve but it is more dramatic for the *k-d* and SA methods. Now the advantages of the greater flexibility in plane location and partitioning dimension make the *k-d* or SA methods much more attractive. The smaller average height of the *k-d* and SA hierarchies(refer to the Spd.Balls and Spd.Tree databases) is another reason for their superior performance. This is especially true for the larger scene models, a very desirable factor since performance is more critical for complex scenes.

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	8.72	99964.82	195.02	3.20
DNA	12.47	31183.64	243.20	6.26
Arches	14.60	153.63	858.33	15.29
Spd.Balls	15.57	568.55	296.89	33.80
Spd.Tree	15.20	2243.42	162.93	18.17
Spd.Mount	14.01	930.41	428.87	30.65

Table 7: SA subdivision, no bounding volumes, *k-d* traversal.

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	11.47	0.07	144.45	2.48
DNA	11.91	723.45	229.10	6.26
Arches	15.79	0.95	304.17	8.17
Spd.Balls	15.74	1.95	260.74	33.32
Spd.Tree	13.77	61.26	155.97	21.40
Spd.Mount	13.91	5.09	272.00	25.98

Table 8: BSP subdivision, bounding volumes, *k-d* traversal.

Tables 5 and 11 show the performance of the BSP scheme with the two different hierarchy traversal methods. Clearly, the *k-d* traversal is superior because of the smaller number of operations done. It is also more general since it can be used in the presence of bounding volumes in the hierarchy.

6 Analysis

From the experiments, we see that there are two very different classes of methods. The BSP, octree (and uniform subdivision, too) belong to one group while the *k-d* and SA methods belong to another. In the BSP/octree schemes, the plane locations are predetermined to bisect the scene extents and the partitioning dimension follows a strict sequence (for the octree, subdivision is simultaneous on all three dimensions). These methods assume objects

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	8.78	960.63	152.55	2.56
DNA	11.14	1363.29	216.40	5.87
Arches	13.72	1.442	320.21	8.02
Spd.Balls	14.97	4.47	259.53	24.60
Spd.Tree	14.02	6.77	125.15	13.50
Spd.Mount	13.97	1617.71	282.45	24.60

Table 9: *k-d* subdivision, bounding volumes, *k-d* traversal.

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	8.72	735.80	144.84	2.23
DNA	10.98	866.59	212.11	5.58
Arches	14.60	1.00	286.01	7.08
Spd.Balls	15.20	3.99	244.51	26.71
Spd.Tree	14.51	5.52	127.55	12.92
Spd.Mount	14.40	4.63	244.63	19.36

Table 10: SA subdivision, bounding volumes, *k-d* traversal.

are uniformly distributed, in which case this is the desirable location for the partitioning plane. Any deviation from this assumption results in an uneven distribution of objects in the individual regions. In the worst case, all objects might fall onto one side of the plane. However, the subdivision is adaptive, in that the empty region is no longer subdivided. Thus, the disadvantage arising from the inflexibility in plane locations is counterbalanced by the adaptive subdivision, which also helps in keeping the void area low.

Both the *k-d* tree and SA methods attempt to locate the partitioning plane in the vicinity of the objects. In the *k-d* hierarchy, the object median is the choice, while in the SA method, the plane is located to minimize the criterion function given in Equation 1. This function takes into account both the number of objects and the surface areas on both sides of the plane. While this is desirable, the heuristics have the potential to create large void spaces

Scene	Avg. Ht	Void Area	flops/ray	Run time
Tetra	11.46	433388.03	249.42	3.46
DNA	11.71	32552.10	339.36	7.62
Arches	15.37	59.12	593.98	12.05
Spd.Balls	20.06	3034.28	1053.52	102.45
Spd.Tree	24.52	48611.76	1146.51	108.58

Table 11: BSP Subdivision, no bounding volumes, BSP Traversal.

on both sides of the plane. Large collections of rays intersect these void spaces and rising traversal costs quickly degrade performance, as shown in Tables 6 and 7.

The void area figures in Tables 5, 6 and 7 help explain the relative performance of these three schemes in the absence of bounding volumes in the hierarchy. The BSP/octree has the least void area and performs best. The introduction of bounding volumes into the hierarchy dramatically improves performance, as indicated by Tables 9 and 10. Now, the best performance is obtained by using the SA heuristic, with bounding volumes at nodes of the hierarchy where they are needed. In this regard, it must be clarified that our implementation of the SA heuristic is different from [17], because of the presence of bounding volumes. At each node of the hierarchy,

1. The object median (equal numbers of object primitives on both sides of the plane) is determined by a binary search within the scene extent.
2. The area median (equal bounding volume surface areas on both sides of the plane) is determined by a second binary search.
3. A linear search is performed between the object and area medians. The plane that best minimizes the criterion function given in Equation 1 is the desired plane choice.

Between the *k-d* and SA methods, the latter performs better because it exploits all the mentioned characteristics critical to performance. The *k-d* hierarchy does not consider surface areas of the regions when trying to choose a partitioning plane. The SA method also solves the problem of disproportionately sized objects in a scene.

From an implementation point of view, the BSP/octree schemes are the easiest to implement, since the other two schemes involve searching for a suitable plane location. Using bounding volumes is definitely an advantage as the figures in Tables 9 and 10 show. The SA heuristic gives the best performance, although the *k-d* hierarchy with bounding volumes comes very close to it.

Lastly, the *k-d* traversal performs significantly better than the BSP/octree traversal and its use is recommended when using the BSP or octree hierarchies.

7 Conclusions

A detailed study of space partitioning techniques has been made with the help of characteristics that are critical to performance. Experimental results clearly support the effects of these characteristics on performance. This has led us to a greater insight and understanding of the performance of some of the most commonly used ray tracing hierarchies.

8 Acknowledgements

Our thanks to Dr. Roy Jenevein and Applied Research Laboratories for use of their computer facilities. Gordon Fossum created the arches model. The Spd.Balls, Spd.Tree and Spd.Mount models were due to Eric Haines [12].

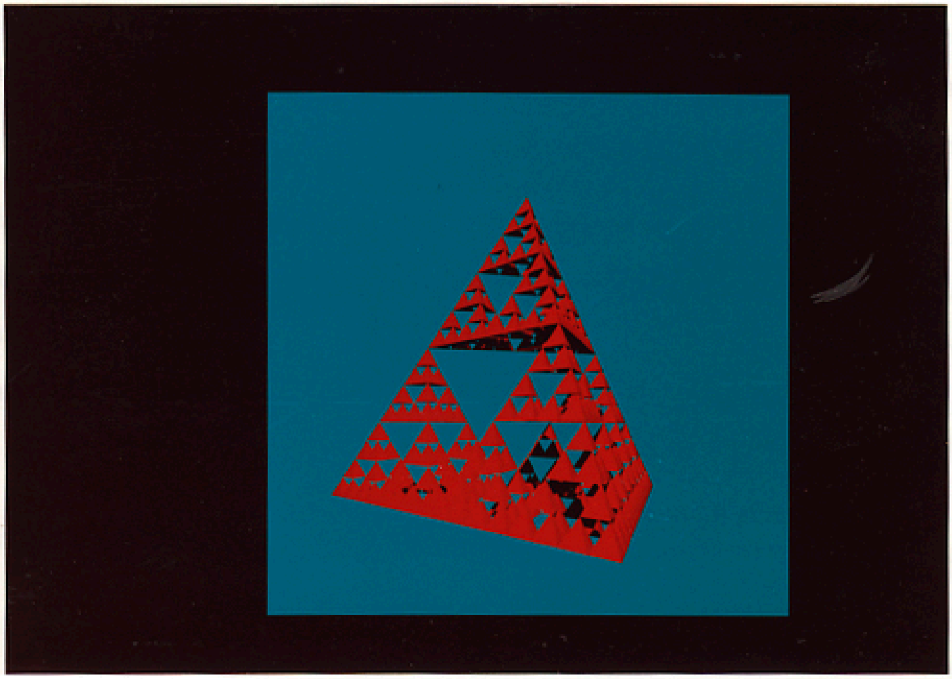

Plate 1. Tetrahedra.

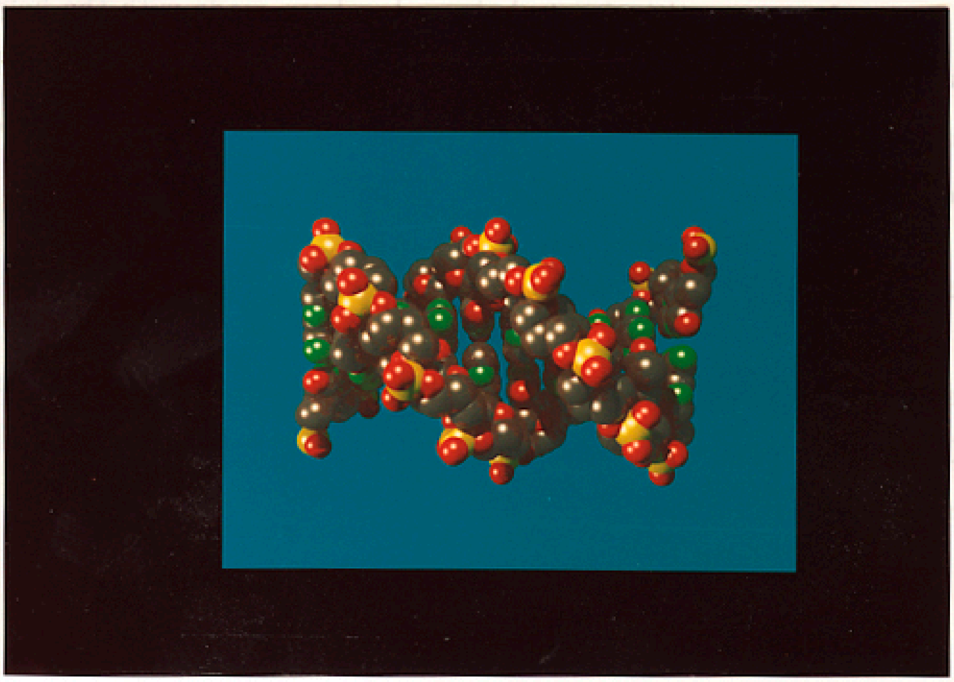

Plate 2. DNA.

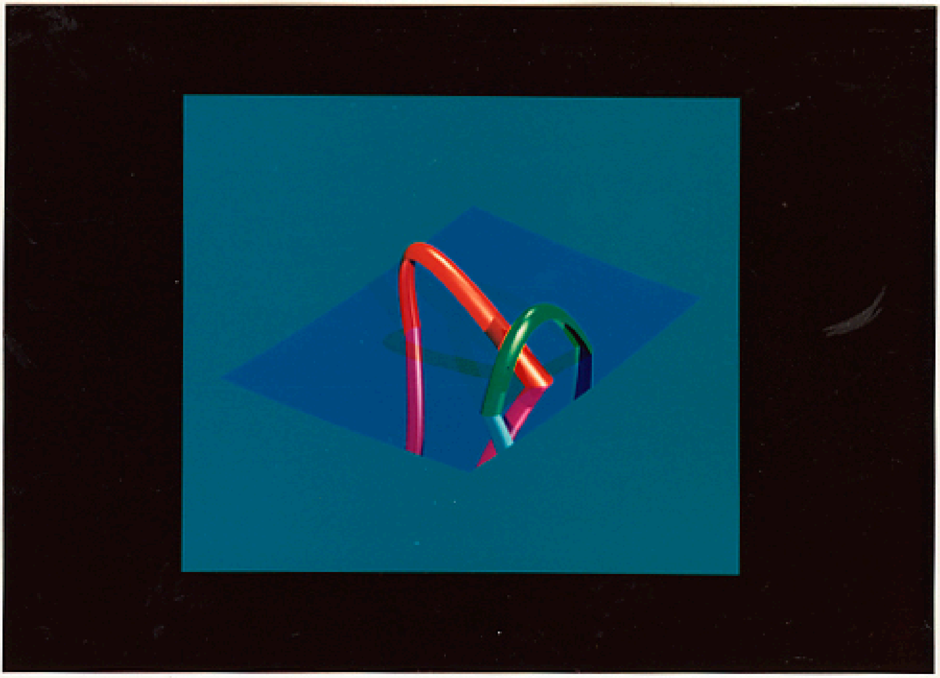

Plate 3. Arches.

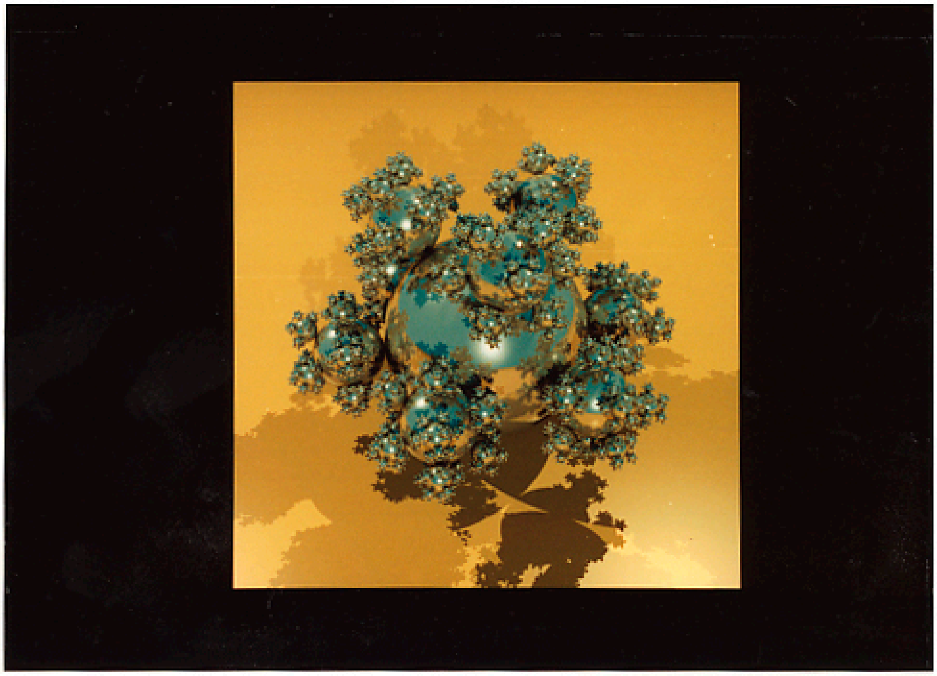

Plate 4. Spd.Balls.

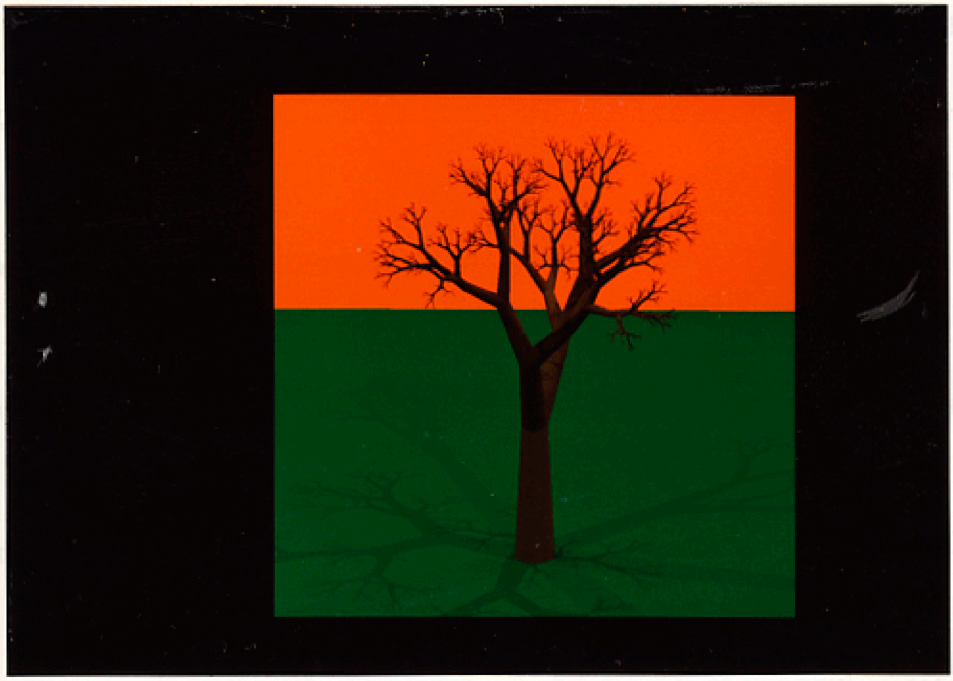
Plate 5. Spd.Tree.

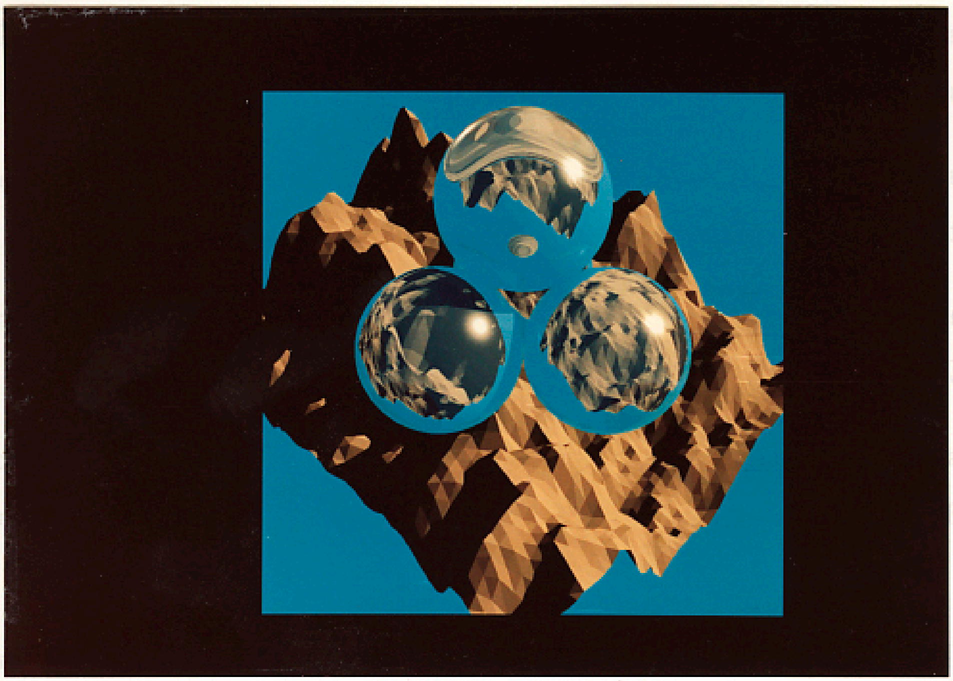
Plate 6. Spd.Mount.

References

[1] James Arvo and David Kirk. Fast ray tracing by ray classification. *Computer Graphics*, 21(4):269–278, July 1987.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), September 1975.

[3] Jon Louis Bentley. Data structures for range searching. *Computing Surveys*, 11(4), December 1979.

[4] John G. Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Visual Computer*, 4(2):65–83, July 1988.

[5] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics*, 18(3):137–145, July 1984.

[6] Henry Fuchs, Gregory D. Abram, and Eric D. Grant. Near real-time shaded display of rigid objects. *Computer Graphics*, 17(3):65–72, July 1983.

[7] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics*, 14(3):124–133, July 1980.

[8] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, April 1986.

[9] Donald Fussell and K. R. Subramanian. Fast ray tracing using k-d trees. Technical Report TR-88-07, Department of Computer Sciences, The University of Texas at Austin, March 1988.

[10] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[11] Jeff Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, pages 14–20, May 1987.

[12] Eric A. Haines. A proposal for standard graphics environments. *IEEE Computer Graphics and Applications*, pages 3–5, November 1987.

[13] Eric A. Haines and Donald P. Greenberg. The light buffer: A shadow testing accelerator. *IEEE Computer Graphics and Applications*, pages 6–16, September 1986.

[14] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. *Computer Graphics*, 18(3):119–127, July 1984.

[15] Michael R. Kaplan. The uses of spatial coherence in ray tracing. *ACM SIGGRAPH Course Notes 11*, July 1985.

[16] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, August 1986.

[17] J. David Macdonald and Kellog S. Booth. Heuristics for ray tracing using space subdivision. *Graphics Interface*, May 1989.

[18] Bruce F. Naylor and William C. Thibault. Application of bsp trees to ray tracing and csg evaluation. Technical Report GIT-ICS-86/03, School of Information and Computer Science, Georgia Institute of Technology, February 1986.

[19] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, 1980.

[20] L. Richard Speer, Tony D. DeRose, and Brian A. Barsky. A theoretical and empirical analysis of coherent ray tracing. *Graphics Interface*, pages 11–25, May 1985.

[21] K. R. Subramanian. Fast ray tracing using k-d trees. Master's thesis, Department of Computer Sciences, The University of Texas at Austin, December 1987.

[22] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterisation of ten hidden-surface algorithms. *Computing Surveys*, 6(1), March 1974.

[23] Kelvin Thompson and Donald S. Fussell. Amalgamated ray tracing, inc. To be submitted.

[24] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.

[25] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.