Ray Tracing Complex Scenes

Timothy L. Kay
James T. Kajiya
California Institute of Technology
Pasadena, CA 91125

Abstract

A new algorithm for speeding up ray-object intersection calculations is presented. Objects are bounded by a new type of extent, which can be made to fit convex hulls arbitrarily tightly. The objects are placed into a hierarchy. A new hierarchy traversal algorithm is presented which is efficient in the sense that objects along the ray are queried in an efficient order.

Results are presented which demonstrate that our technique is several times faster than other published algorithms. Furthermore, we demonstrate that it is currently possible to ray trace scenes containing hundreds of thousands of objects.

Keywords: Ray tracing, extent, bounding volume, hierarchy, traversal

1 Introduction

Our purpose in studying ray tracing is to develop techniques for synthesizing complex, realistic images. We have two primary requirements; the rendering program must produce images of the highest possible quality, and it must be able to handle hundreds of thousands or millions of objects.

Rubin and Whitted [Rubin 80] determined that their program spent most of its time computing ray-object intersections in order to calculate which object was visible for a given ray. The performance worsened linearly as the number of objects in the scene increased. To improve performance, they proposed placing simple bounding volumes around each object in their database. If a given ray failed to intersect the bounding volume for a particular object, the object needed no further consideration. This way, they avoided many ray-object intersection calculations. By grouping objects hierarchically and placing a bounding volume encompassing the extent of all children at each node of the hierarchy tree, the majority of the ray-bounding volume intersection calculations could also be avoided. Rubin and Whitted chose rectangular solids as bounding volumes. To improve tightness, they deformed the rectangular solids with an affine matrix, thereby requiring a matrix-vector multiply before the ray-bounding volume test could be performed.

Weghorst, et. al. [Weghorst 84] proposed and benchmarked a similar scheme. Notably, they studied the use of different types of bounding volumes in a single hierarchy.

Concurrently but independently, Glassner [Glassner 84] and Kaplan [Kaplan 85] and more recently Fujimoto, et. al. [Fujimoto 86] studied a different approach to the problem. Whereas Rubin and Whitted and Weghorst created bounding volumes to surround their objects, Glassner and Kaplan divided their objects into cells of an octree. As a ray passed through space, the octree data structure rejected any objects trivially far away from the ray. A significant feature of the Glassner-Kaplan algorithm is that it considered objects along a given ray roughly *in the order that they occurred along the ray.* A major drawback of this space-partitioning approach is that any particular object might end up in more than one octree cell, potentially requiring a ray to be intersected with the same object more than once.

In this paper we partition objects rather than space. We present a new type of bounding volume for which a ray intersection requires very little computation. Unlike previously described bounding volumes, ours have the advantage that they can be made to fit the convex hull of an object arbitrarily tightly in exchange for a slower intersection computation.

We present a new algorithm for traversing a hierarchy of such bounding volumes. The algorithm is efficient in the sense that, for a given type of bounding volume and a given ray, the objects are queried in a well-defined, efficient order regardless of the hierarchy.

The results section compares our method to recently-published work and finds that our algorithm is roughly three times faster. More importantly, our technique has proven capable of rendering scenes containing over one hundred thousand objects. This enabled us to compute two sequences for the SIGGRAPH '84 Omnimax film *The Magic Egg* and the piece *Trees* in the SIGGRAPH '85 film show.

2 Object Extents

The *extent* of an object is the region of space occupied by the object. For the sake of computational simplicity, we wish to bound each object in a volume that is simpler than the object itself. If a ray-extent intersection is significantly faster than a ray-object intersection, the results of the ray-extent intersections can be used to prune a majority of the expensive ray-object intersections. If a ray does not hit an object's bounding volume, then clearly it will miss the object itself.

In choosing bounding volumes for arbitrarily complex objects, two opposing constraints must be balanced. If a bounding volume fits an object loosely, many rays that hit the bounding volume will miss the object. If a bounding volume describes an object's extent precisely, very few ray-object intersection calculations will be wasted. By this criterion alone, the best bounding volume for an object is the object itself. On the other hand, intersecting rays with simpler bounding volumes requires fewer calculations. This constraint suggests that simple spheres, ellipsoids and rectangular solids are good bounding volumes. For example, Rubin and Whitted and Toth [Toth 85] used rectangular solids to describe object extents, while Weghorst used a mixture of objects such as rectangular solids and spheres.

With respect to the tightness versus speed tradeoff, the bounding volumes now presented can be tailored to suit the particular scene being rendered.

2.1 Bounding Volume Description

We bound objects with parallelopipeds constructed of planes. An arbitrary plane in 3-space is described implicitly by the equation

$$Ax + By + Cz - d = 0. \qquad (1)$$

Geometrically, Equation 1 describes a plane with normal vector $\hat{P}_i = \begin{pmatrix} A \\ B \\ C \end{pmatrix}$ lying d units from the origin.

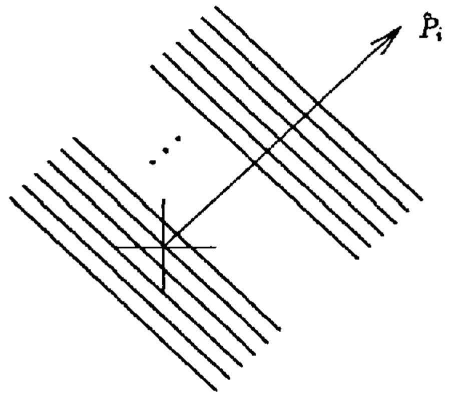

Figure 1: The set of planes defined by $\hat{P}_i$

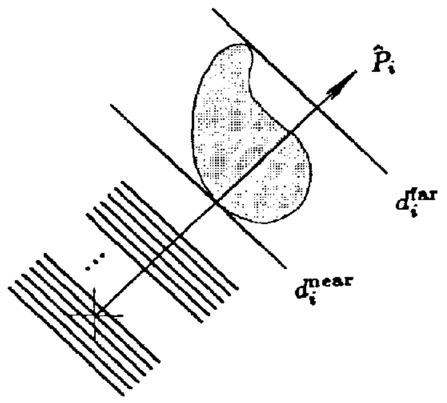

Figure 2: Planes corresponding to $\hat{P}_i$ that bound an object

If the normal vector $\hat{P}_i$ is fixed, leaving the distance d free to vary, Equation 1 then describes the set of all planes normal to $\hat{P}_i$. See Figure 1. Out of any such set we choose two planes that bound a given object. See Figure 2. Significantly, we can describe these two planes by exactly two real numbers d_i^{near} and d_i^{far}, the values of d for the two planes. We call the region in space between such planes a *slab*. The normal vector defining the orientation of a slab is termed a *plane-set normal.*

Different choices of plane-set normals $\hat{P}_i$ yield different bounding slabs for an object. The intersection of a set of bounding slabs yields a bounding volume. In order to create a closed bounding volume in 3-space, at least three bounding slabs must be involved, and they must be chosen so that the defining normal vectors span 3-space. For the sake of simplicity, the examples in the figures are presented in 2-space, in which a minimum of two vectors will span. The generalization to 3-space requires the addition of a third coordinate to each vector; the equations generalize without modification. Figure 3a shows an object bounded by slabs with nor-

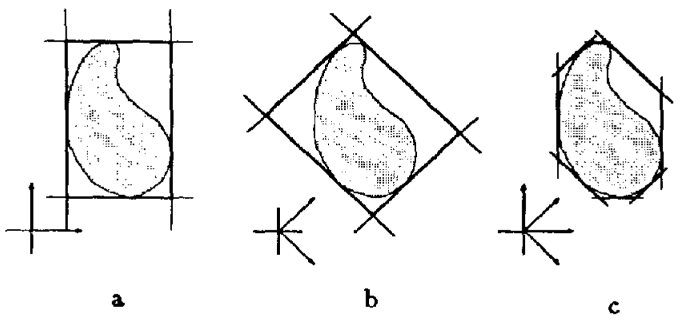

Figure 3: Bounding an object using various normals

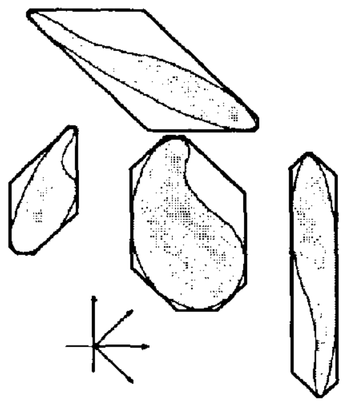

Figure 4: Objects bounded by a fixed set of normals

mals $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Figure 3b shows the same object bounded using normals $\begin{pmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{pmatrix}$ and $\begin{pmatrix} \frac{\sqrt{2}}{2} \\ \frac{-\sqrt{2}}{2} \end{pmatrix}$. Figure 3c shows the same object bounded using all four normals.

A realistic scene will contain millions of objects. Storing plane-set normals and corresponding d^{near} and d^{far} for each object would require a great deal of memory and, as we will soon see, a great deal of computation. Therefore, we choose our plane-set normals in advance, independent of the particular objects to be bounded. This will restrict our choices of bounding volumes, but will allow us to describe them by a small set of numbers. Figure 4 shows several objects bounded by volumes restricted to preselected plane-set normals.

The plane-set normals $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ define rectangular solid bounding volumes whose faces are perpendicular to the coordinate axes. The plane-set normals $\begin{pmatrix} \frac{\sqrt{3}}{3} \\ \frac{\sqrt{3}}{3} \\ \frac{\sqrt{3}}{3} \end{pmatrix}$, $\begin{pmatrix} \frac{-\sqrt{3}}{3} \\ \frac{\sqrt{3}}{3} \\ \frac{\sqrt{3}}{3} \end{pmatrix}$, $\begin{pmatrix} \frac{-\sqrt{3}}{3} \\ \frac{-\sqrt{3}}{3} \\ \frac{\sqrt{3}}{3} \end{pmatrix}$ and $\begin{pmatrix} \frac{\sqrt{3}}{3} \\ \frac{-\sqrt{3}}{3} \\ \frac{\sqrt{3}}{3} \end{pmatrix}$ define eight-sided parallelopipeds.

Bounding volumes constructed using a larger number of plane-set normals will bound objects more tightly. For example, Figure 13 shows a tree for which we wish to compute bounding volumes. Figure 8a shows the bounding volumes created using the first three normals mentioned in the previous paragraph. Figure 8b shows the volumes formed using all seven normals.

2.2 Computing Bounding Volumes

The d_i^{near} and d_i^{far} values are simply the endpoints of the extent of the projection of the object onto the normal $\hat{P}_i$

A modeling transformation is associated with each object in the database. This transformation must be taken into account because the bounding volumes exist in world space. We will treat a modeling transformation as a three-by-three matrix M and a translation vector T. We describe the bounding volume calculation for several types of objects.

Example 1. Polyhedra

A polyhedron is defined by a collection of vertices $\begin{pmatrix} x_j \\ y_j \\ z_j \end{pmatrix}$. We compute the bounding volume for the polyhedron by computing the bounding volume enclosing these vertices. There are three steps.

1. The bounding volume belongs in final world coordinates, the vertices must be transformed by M.

$$\begin{pmatrix} x'_j \\ y'_j \\ z'_j \end{pmatrix} = M \begin{pmatrix} x_j \\ y_j \\ z_j \end{pmatrix} + T.$$

2. Each vertex is projected onto each $\hat{P}_i$.

$$d_{ij} = \left(\hat{P}_i\right)^{\text{T}} \begin{pmatrix} x'_j \\ y'_j \\ z'_j \end{pmatrix}.[1]$$

3. Corresponding to each $\hat{P}_i$, compute

$$d_i^{\text{near}} = \min\{d_{ij}\}$$

and

$$d_i^{\text{far}} = \max\{d_{ij}\}.$$

[1]A row vector followed by a column vector denotes dot product.

Example 2. Implicit surfaces

Spheres and other implicit surfaces must be approached differently. We wish to compute bounds for the set of points $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ which satisfy the implicit equation of the surface. We will first treat the transformation by M and then account for T afterwards.

After transforming by M, we project each point onto $\hat{P}_i$. The transformation and projection yield

$$f\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \hat{P}_i^{\mathrm{T}}\left(M\begin{pmatrix} x \\ y \\ z \end{pmatrix}\right).$$

We apply the associative property.

$$f\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \left(\hat{P}_i^{\mathrm{T}} M\right)\begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

If we let

$$\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \left(\hat{P}_i^{\mathrm{T}} M\right)^{\mathrm{T}} = M^{\mathrm{T}}\hat{P}_i,$$

we can rewrite f as

$$f\begin{pmatrix} x \\ y \\ z \end{pmatrix} = Ax + By + Cz. \tag{2}$$

Here we have a scalar field constrained by the implicit equation, and we wish to find the minimum and maximum values assumed by f. Problems of this type can be solved readily using the method of Lagrange multipliers (see [Apostol 69]).

Next we account for the translation T. We need to translate each $(d_i^{\mathrm{near}}, d_i^{\mathrm{far}})$ pair by the length of the component of T parallel to $\hat{P}_i$, which is simply $T \cdot \hat{P}_i$.

For example, the implicit equation of a sphere is

$$x^2 + y^2 + z^2 - 1 = 0. \tag{3}$$

The method of Lagrange multipliers tells us that the extrema occur when

$$\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \lambda\begin{pmatrix} 2x \\ 2y \\ 2z \end{pmatrix}.$$

Substituting back into Equations 2 and 3 gives

$$f\begin{pmatrix} x \\ y \\ z \end{pmatrix}\Bigg|_{x^2+y^2+z^2-1=0} = \pm\sqrt{A^2 + B^2 + C^2}.$$

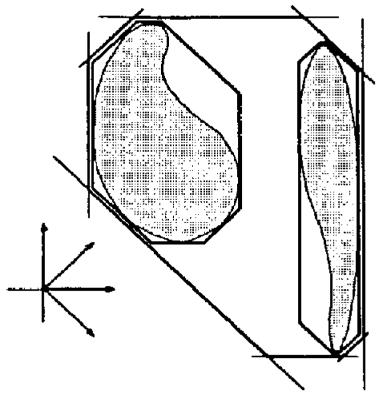

Figure 5: The bounding volume of two bounding volumes

The contribution of T to the position of each slab is $T \cdot \hat{P}_i$. Combining the contributions from M and T yields

$$(d_i^{\mathrm{near}}, d_i^{\mathrm{far}}) = \left(T \cdot \hat{P}_i - \left|M^{\mathrm{T}}\hat{P}_i\right|, T \cdot \hat{P}_i + \left|M^{\mathrm{T}}\hat{P}_i\right|\right).$$

Example 3. Compound objects

The bounding volumes for compound objects such as booleans (constructed as unions, intersections and subtractions of other objects) can be calculated by first computing bounding volumes for each subobject. These bounding volumes can then be combined. As Figure 5 demonstrates, the bounding volume of two bounding volumes is simply the pairwise minimum of each d_i^{near} and the pairwise maximum of each d_i^{far} value.

For example, the branches of the trees in the results section are cylinders with a sphere at each end. A branch's bounding volumes is computed as a combination of the two sphere's bounding volumes. We can ignore the bounding volume of the cylinder because it is guaranteed to lie within the combined bounding volume of the two spheres.

2.3 Ray-Volume Intersection

The intersection algorithm presented here is similar in spirit to that of Cyrus and Beck [Cyrus 78].

A ray intersects with a slab to yield an interval along the ray. To compute this interval, we intersect the ray

$$R = \hat{a}t + b \tag{4}$$

with the each of the two planes bounding the slab. The solution of this intersection in terms of the ray parameter t is computed by substituting the ray Equation 4 into the plane Equation 1 yielding

$$t = \frac{d_i - \hat{P}_i \cdot b}{\hat{P}_i \cdot \hat{a}}. \tag{5}$$

The computation for both planes is identical, and each provides one endpoint to the interval along the ray. Care must be taken to orient the two endpoints correctly. If the denominator of Equation 5 is less than zero, then the roles of the near and far values must be reversed. Also, in the event that this dot product is close to zero, the division in Equation 5 might cause an overflow (or division by zero). We will provide a solution to this problem later.

To compute the intersection of a ray with a bounding volume, we compute the intersection of the ray with each slab, and then compute the intersection of these intervals. We compute the intersection of the intervals by computing the maximum of the near values and the minimum of the far values.

If the ray happened to miss the bounding volume, then t^{near} will be larger than t^{far}. Otherwise we have the value of t at the two intersections of the ray with the bounding volume. These values are useful as estimates of the position of the object along the ray.

2.4 Cost of Intersection

By counting operations in Equation 5 we see that each ray-slab intersection requires four dot products, two subtracts and two divides when taking both planes into account. These numbers must be multiplied by the number of slabs involved in each bounding volume, and a minimization and a maximization must be done across the slabs.

But we have determined our choice of plane-set normals *a priori*. For a given ray, we can compute the dot products to be used in all calculations *just once*. If there are many ray-bounding volume intersections associated with each ray, and this is generally the case, the expense of the dot product computation will be of little significance. Furthermore, at the time that the dot products are precomputed, we can calculate the reciprocal of the denominator of Equation 5 and replace the divide with a multiply. Before taking its reciprocal, though, the value must be checked for the possibility of overflow or divide by zero. If this is the case, a large number can be used in place of the reciprocal. [2]

$$t = (d_i - S)T.$$

where $S = \hat{P}_i \cdot b$ and $T = \frac{1}{\hat{P}_i \cdot \hat{a}}$.

The new computation requires two subtracts, two multiplies and a comparison for each slab contributing to the bounding volume. The calculation is now very fast.

[2] Let M be the square root of the largest number representable by the machine. If we restrict the database to lie within a sphere of this radius, centered at the origin, then we can safely replace the reciprocal with M and avoid overflows.

The expense of a ray-bounding volume intersection grows linearly with the number of slabs used to bound an object. A slight modification improves performance. Most rays miss most bounding volumes by a wide margin. When this occurs, the intersection of the different ray-slab computations will be empty after no more than three slabs are processed. Therefore, the interval intersection calculation should be done in tandem with the ray-slab intersection calculation. If the intersection becomes empty, the rest of the ray-slab intersections can be ignored.

3 Object Hierarchies

While a large savings can be obtained by placing each object in a bounding volume, the cost of computing an image remains proportional to the number of objects in the image. A much larger savings can be achieved by creating a hierarchy of objects. We define the extent of a particular node in the hierarchy to be the combined extent of all that node's children. For each such node we compute a bounding volume. Then if a ray misses the bounding volume at a given node in the hierarchy, we can *reject that node's entire subtree from further consideration*. Using a hierarchy causes the cost of computing an image to behave logarithmically in the number of objects in the scene.

In creating a hierarchy, we require one fundamental operation on bounding volumes. We must be able to compute the bounding volume of a particular node in the hierarchy tree given the bounding volumes of that node's children. The volume enclosing two other bounding volumes is simply the minimum of the d_i^{near} and the maximum of the d_i^{far} values for the pair of bounding volumes in question.

3.1 Hierarchies

There are many schemes for constructing a hierarchy. Some are easy to implement and fast to execute, while others might be sophisticated and consume large amounts of computer time. There are several interrelated properties that would seem to distinguish a good hierarchy from a bad one.

- Any given subtree should contain objects that are near each other. "Nearness" is relative, but the lower the subtree is with respect to the entire hierarchy, the "nearer" objects should be.

- The volume of each node should be minimal.

- The sum of the volume of all bounding volumes should be minimal.

- The construction of the tree should concentrate on the nodes nearer the root of the tree. Pruning a branch of the tree there allows a large subtree to be remove from further consideration, whereas pruning one lower down removes just a few bounding volumes and objects from further consideration.

- The time spent constructing the hierarchy tree should more than pay for itself in time saved rendering the image.

The simplest hierarchy involves almost no computation, and gives no consideration to the "nearness" criterion. Taking the objects in the order that they are described, we simply create a tree with a fixed branching ratio using a bottom-up construction. This construction does moderately well with regard to nearness if the database is modeled in a coherent fashion.

We could spend additional computer time to construct a better hierarchy. A *median-cut* scheme will help group near objects together even if they aren't modeled coherently. The scheme constructs a binary tree in a top-down fashion. At each level all the objects in the database are sorted by their x coordinate. The objects are then partitioned at their median. This defines which objects belong on each side of the tree. The process is repeated recursively, except that at each level the objects are sorted and partitioned on a different coordinate axis.

The previous algorithm attempts to group objects based on their nearness to each other, but it does so along one dimension at a time. It might do better to group objects in along three dimensions at once by employing an octree. This preprocessing step would be very similar to the one described in [Glassner 84] or [Kaplan 85]. The difference is that, if an object straddles two or more octree cells, it is arbitrarily placed in exactly one of them. Furthermore, after the preprocessing is completed, the octree is used solely to define the hierarchy; each object must still be assigned a distinct bounding volume.

4 Hierarchy Traversal

We present a traversal algorithm whose output is insensitive to the manner in which the hierarchy is constructed. This traversal algorithm bears similarities to the fractal intersection algorithm of Kajiya [Kajiya 83]. We will assume that the type of bounding volume to be used has has been fixed *a priori*. We demonstrate that for any ray there is a preferred order with which to query the objects in the database.

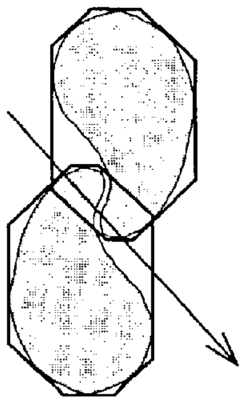
Figure 6: The ray tracing program must compute two object intersections

The intersection of a ray with an object's bounding volume yields two numbers. We call the nearer number the *estimated distance* of the object along the ray. For a given ray, a bounding volumes will assign a unique estimated distance to each object in the database. These numbers impose a total order on the objects which defines the best order to search the database to resolve the ray's intersection.

The first object in the preferred order is not necessarily the one in which we are interested. Figure 6 demonstrates that the first object is not always the visible object. The best we can do is to proceed through the objects in order and compute ray-object intersections. If we find an object that intersects the ray at a point closer than the remaining objects' distance estimates, then the object in question is the visible object.

The hierarchy traversal algorithm presented in Figure 7 will search the database in the total order just defined. Because objects are queried in the same order regardless of the object bounding volume hierarchy, the difference between a poorly constructed hierarchy and a good one is slight. Given a poor hierarchy, the program will spend more time computing ray-bounding volume intersections. *The performance of the rest of the program (ray-object intersections, shading calculations) will be unaffected.*

The traversal algorithm searches the hierarchy tree, but unlike a depth first or breadth first search, it traverses the hierarchy tree in an order derived while the traversal is in progress. A *candidate* node is one whose bounding volume is known to be in the path of the ray, but whose children have yet to be interrogated. We keep track of candidate nodes in a priority queue, implemented as a heap (see [Aho 74] or [Sedgewick 83]). The heap guides the search by selecting the candidate with the smallest nonnegative distance estimate.

We have found that the use of a heap is essential to

```
We are given a ray
Compute dot products and reciprocals (section 2.4)
Let $t$ = distance to the nearest object hit so far
Initialize $t = +\inf$
Let $p$ contain a pointer to the nearest object hit so far
Initialize $p$ = nil
While heap is non-empty and distance to top node $< t$
 Extract candidate from heap
 If the candidate is a leaf
   Compute ray-object intersection
   If ray hits candidate and distance $< t$ then
     $t$ = distance
     $p$ = candidate
   Endif
 Else
   For each child of the candidate
     Compute ray-bounding volume intersection
     If the ray hits the bounding volume
       Insert the child into the heap
     Endif
   Endfor
 Endif
Endwhile
```

Figure 7: Traversal algorithm

the efficiency of the algorithm. In previous implementations it was found that a tremendous amount of time was spent in sorting routines. A heap does a better job while consuming much less computer time.

5 Results

The ray tracing scheme just described has been under development for several years. A primitive version was used to compute the *Space Colony* and *Saturn Flyby* sequences from the SIGGRAPH '84 Omnimax film *The Magic Egg*.

A revised version of the program was used to compute *Trees*, which was shown at the SIGGRAPH '85 Film Show. Figure 9 is a frame from this movie. The *Trees* database contains about 110,000 objects.

The ability to compute these two movies demonstrates the feasibility of our algorithms. We expect to ray trace a scene containing at least one million objects by the end of 1986.

The test images represent an increasing sequence of scene compositions and complexities. With Figure 10 we attempt to duplicate a test case in Glassner's paper. The pyramid is constructed of 1024 triangles. Figure 11 contains four copies of the pyramid from Figure 10. Figure 12 contains 2 superquadrics [Barr 81] and 400 tiles each composed of 6 faces. Figure 13 contains a base polygon and a single tree composed of 1271 branches. Figure 14 contains the same objects as Figure 13 in addition to 6184 polygonal leaves.

	Pyramid ** 4	Pyramid ** 5	Super-quadric	Tree Branches	Tree Leaves	Trees
Objects	1024	4096	2402	1272	7455	110000+
Lights	1	1	2	1	1	1
Pixels	262	262	262	262	262	262
Color Rays	263	263	263	263	263	263
Shadow Rays	37	34	410	135	155	
Total Rays	300	297	673	398	418	
Rays that Hit	43	41	241	151	206	
Object Int.	521	330	841	364	983	
B. V. Int.	2810	3678	44802	18491	33783	288000@
Time 4381 sec	980	1033	5147	2429	4116	
(sec/ray)	.00327	.00348	.00765	.00610	.00985	
(sec/ray that hit)	.02276	.02523	.02138	.01608	.01998	

All pixel, ray and intersection counts are in thousands.

@ Computed on a 3081 using an early version of the code.

Table 1: Timing data

All images were computed on an IBM 4381/Group 12 using Amdahl UTS, a version of Unix/System V. Depending on the scene, a 4381 benchmarks 3.5-4.0 times faster than a Vax 11/780 running Berkeley Unix 4.2. The pictures were computed at 512 by 512 pixel resolution. For the sake of the presentation, all images in this publication were antialiased using 4 by 4 subsampling. The samples were then filtered using an eight-lobe, windowed sinc function. The values reported in Table 1 correspond to the same images computed with one sample per pixel.

It is reasonable to present the benchmarks in terms of atomic actions rather than as execution times for pictures of some arbitrary size. Since we are concentrating on the ray casting aspects of ray tracing, we report the speed of the program in units of seconds per ray.

Table 1 contains the statistics for the test images. Timings are given in seconds, seconds per ray, and seconds per object-hitting ray. The time per ray value changes drastically with each image. When the amount of visible background increases, there are more rays that miss the world. These rays require very little computation and thereby bias the statistics. We ignore these rays by keeping track of the object-hitting rays. While the time per ray changes a great deal from image to image, the time per object-hitting ray stays relatively constant.

In their recent papers, Glassner and Weghorst report execution times for their ray tracing programs. We had hoped to do an in-depth comparison of the speeds of their implementations with ours. Despite our efforts, we have found that there are too many unknown factors involved. In the hopes of establishing a basis for future comparison, we offer our databases to anyone who wishes to put their programs through its paces. Furthermore, we welcome discussions regarding which benchmarks would be most informative.

The few comparisons we can make between Glassner's data and our own seem to suggest that our method is significantly faster. We managed to faithfully reproduce only one of his models. We recomputed our version on a Vax/780 for the sake of comparison. Glassner program used 8700 seconds of Vax/780 time to render his "recursive pyramid" of Figure 10, while our implementation took 2706 seconds. Because the viewpoints differed slightly, he shot 352322 rays to our 298588. Taking this into account, we can conclude that, for this particular image, our program runs 2.6 times faster than his.

We can increase the number of plane-set normals beyond the required three. Figure 8 demonstrates that objects can be bounded very tightly in this fashion. The pyramid benchmark ran the fastest when the combined normals of an octahedron were combined with a single additional normal pointing upwards.

When ray tracing on a supercomputer, a larger number of plane-set normals is very advantageous. In that environment, the traversal algorithm may be vectorized, and the incremental cost of additional plane sets is minimal.

6 Conclusion

As databases become increasingly complex, the time spent computing the visible object for a ray becomes the dominant concern. We have presented a new algorithm for speeding up this process.

7 References

[Apostol 69] Apostol, Tom M., *Calculus*, Volume II, Wiley, New York, 1969, pp. 314-318.

[Barr 81] Barr, Alan H., "Superquadrics and Angle Preserving Transformations," *Computer Graphics and Applications*, 1(1).

[Cook 84] Cook, Robert L., Thomas Porter and Loren Carpenter, "Distributed Ray Tracing," Computer Graphics, 18(3), July 1984, pp. 137-145.

[Cyrus 78] Cyrus, M. and J. Beck, "Generalized two and three dimensional Clipping," Computers and Graphics, 3(1), 1978, pp. 23-28.

[Fujimoto 86] Fujimoto, Akira, Takayuki Tanaka, and Kansei Iwata, "ARTS: Accelerated Ray-Tracing System", IEEE Computer Graphics and Applications, 6(4), April 1986, 16-26.

[Glassner 84] Glassner, Andrew S., "Space Subdivision for Fast Ray Tracing," IEEE Computer Graphics and Applications, 4(10), October, 1984, pp. 15-22.

[Kaplan 85] Kaplan, Michael R., "The Uses of Spatial Coherence in Ray Tracing," ACM SIGGRAPH '85 Course Notes 11, July 22-26 1985.

[Kay 86] Kay, Timothy L., M.S. dissertation in preparation.

[Kajiya 83] Kajiya, James T., "New Techniques for Ray Tracing Procedurally Defined Objects", Computer Graphics, 17(3), July, 1983, pp. 91-102.

[Rubin 80] Rubin, Steve M. and T. Whitted., "A Three-Dimensional Representation for Fast Rendering of Complex Scenes," Computer Graphics 14(3), July 1980, pp. 110-116.

[Sedgewick 83] Sedgewick, Robert, *Algorithms*, Addison-Wesley, Reading, 1983, pp. 127-142.

[Toth 85] Toth, Daniel L., "On Ray Tracing Parametric Surfaces," Computer Graphics 19(3), July 1985, pp. 171-179.

[Weghorst 84] Weghorst, Hank, Gary Hooper, and Donald P. Greenberg, "Improved Computational Methods for Ray Tracing," ACM Transactions on Graphics, 3(1), January 1984, pp. 52-69.

[Whitted 80] Whitted, Turner, "An Improved Illumination Model for Shaded Display," Communications of the ACM, 23(6), June 1980, 343-349.

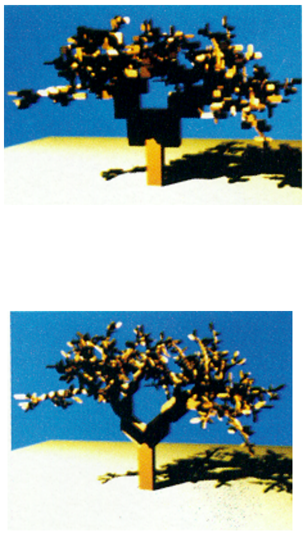
Figure 8b

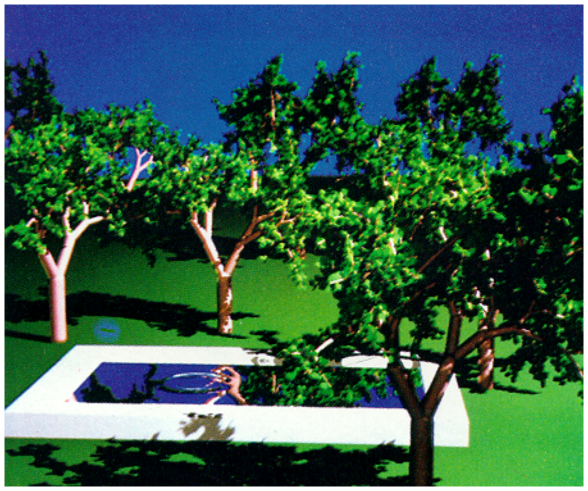
Figure 9

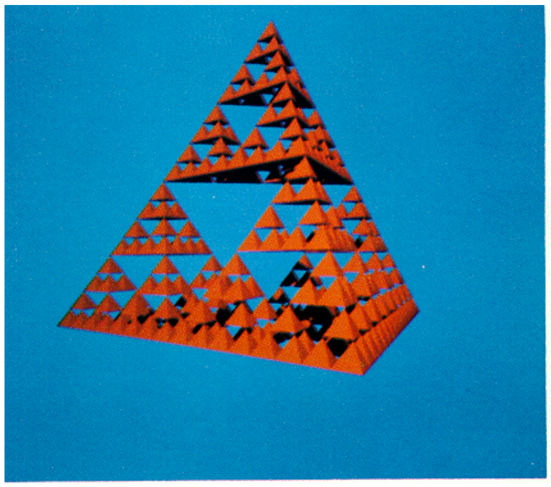
Figure 10

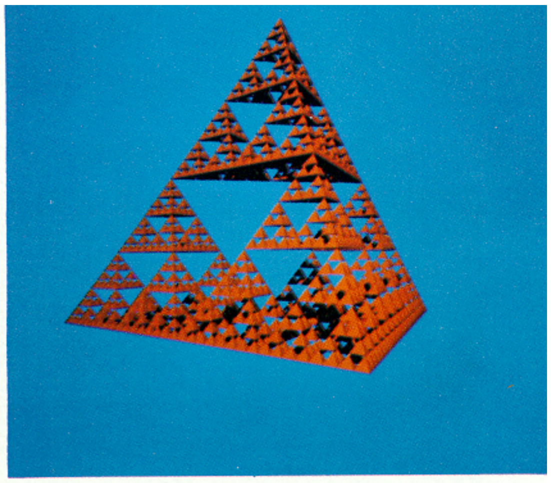
Figure 11

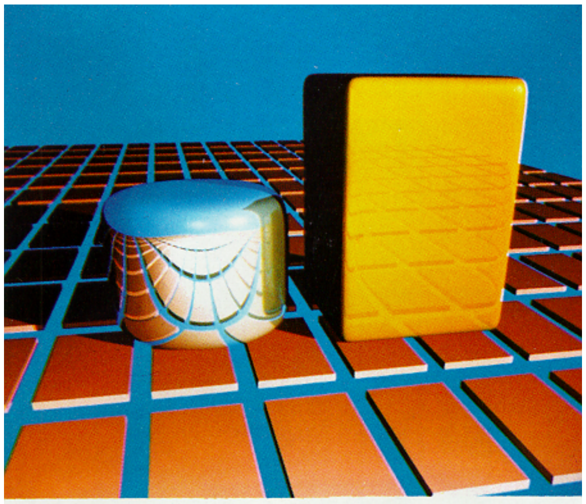
Figure 12

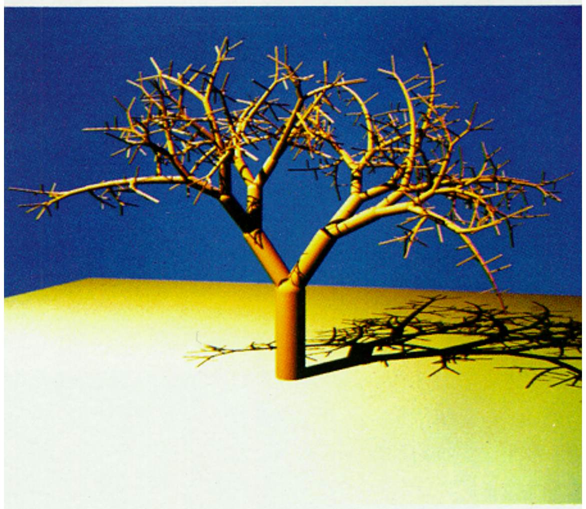
Figure 13

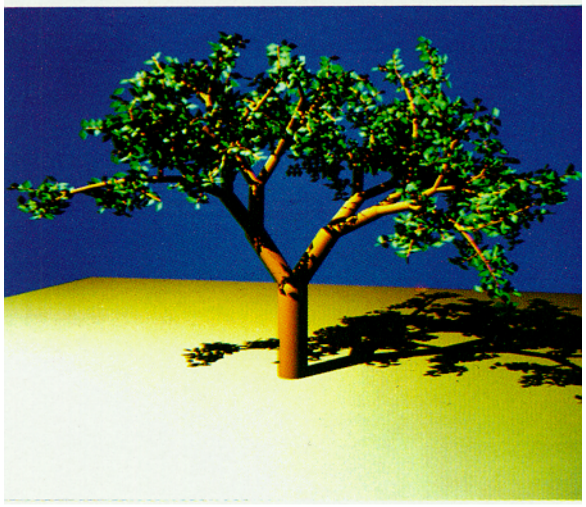
Figure 14