FAST RAY TRACING USING K-D TREES

Donald Fussell and K. R. Subramanian

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

TR-88-07 March 1988

Fast Ray Tracing Using K-D Trees

Donald Fussell
K. R. Subramanian
Department of Computer Sciences
The University of Texas at Austin

Abstract

A hierarchical search structure for ray tracing based on *k-d* trees is introduced. This data structure can handle the variety of surfaces commonly used in computer graphics. Algorithms to build and traverse this binary data structure are described. Only regions through which a ray travels are interrogated and the search proceeds along the path of the ray starting from its origin. The algorithm also ensures that no object is interrogated more than once in the search for intersection. Benchmarking results indicate that when used with axis-aligned bounding parallelepipeds, this method of ray tracing is one of the fastest known.

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, separating plane, hierarchy, traversal.

1 Introduction

In recent years, realistic image generation has been a major focus of research in computer graphics. Ray tracing has established itself as an important means of achieving this goal.

Its ability to determine visible surfaces, compute shadows, model reflection and transmission of light and a host of other effects [4] has made it an important rendering technique. Its major difficulty has been the tremendous amount of computing needed to produce photorealistic images. Given the high cost of producing still frames, animation has been next to impossible with this technique.

The principal expense in ray tracing is determining the first intersection of a ray with an object in a scene. This process has to be done several times per pixel, depending on the effects we are trying to model (reflection, transmission, shadows, penumbras etc). Considerable effort has gone into making this process very efficient. Bounding volumes [18][22][23][14], search structures [8][9][13], item buffers [22], shadow buffers [11] and ray coherence [1][12][20] all have drastically reduced this cost.

Bounding volumes are advantageous because they provide a quick non-intersection test for objects. Using tight bounding volumes improves the chances of intersecting the object inside the volume, but then increases the bounding volume intersection cost. Data structures which support efficient geometric search allow us to look at only a small percentage of the scene to determine the closest intersection. An octree is an example of such a hierarchical data structure. The use of bounding volumes and geometric search structures together allows large sections of space to be quickly eliminated from consideration for any particular ray-object intersection. While many of these techniques are applicable to all stages of the ray-tracing process, some specialized methods have been devised to speed up particularly important subtasks. Item buffers speed up the primary ray intersections, while shadow buffers speed up the shadow intersection calculations. Ray-to-ray coherence is another approach to speed up the intersection search [20][12]. Recent work combining ray-to-ray coherence and novel search structure techniques has proved to be very successful. In this method [1], objects are classified into beams in five-dimensional space with the

ray direction contributing the two additional dimensions.

In this paper, we present a space partitioning technique using the *k-d* tree, a multidimensional geometric search structure introduced by Bentley [3]. Objects in the scene are partitioned along the X, Y and Z dimensions. In order to obtain a balanced structure, the dimension chosen at each partitioning step is decided by the evenness of the partition that can be obtained from the input scene rather than being fixed *a priori* as in traditional *k-d* trees. This search structure is effective because it combines the efficiencies in geometric searching gained from such data structures as octrees with the flexibility to adapt itself to scene characteristics that characterizes hierarchical extent approaches. We present algorithms for building this structure and for traversing it along the path of the ray being traced, starting from its origin. Results of using this data structure on some example scenes show that it is among the most efficient currently known, and that it seems particularly well suited to handling complex scenes by virtue of the relatively small number of objects queried in its search for the closest intersection.

2 K-D Trees and Relevant Work

The multidimensional binary search tree (or the *k-d* tree) was proposed by Bentley [2][3]. It is a generalization of the simple binary search tree to k dimensions. The partitioning dimension at each node of the tree is indicated by a discriminator D, an integer between 0 and $k-1$. At any level l it is given by

$$D = l \ modulo \ k$$

assuming the root node is at level 0. The exact semantics of D will be determined by the application to which the *k-d* tree is put to use.

For space partitioning, we use $k = 3$ (X, Y and Z dimensions), $D \epsilon$ (0, 1, 2) and D at any node of the tree will be determined by the best partition that can be obtained from

the subspace of objects under that node. Here D will be an index representing planes orthogonal to the three coordinate axes.

The work of Shumacker [17][19][21] and Fuchs, Naylor et al. [6][7][16] on the solution to the hidden surface problem is related to our approach to efficient ray tracing using the *k-d* tree. Hence we will briefly describe their work.

Shumacker's algorithm partitions the environment into a set of clusters using hand-picked separating planes. For this, the objects in the environment must be *linearly separable,* i.e., there must exist a plane which partitions the objects into two nonempty sets without intersecting any of the objects. With the environment divided into two subsets, each of these sets is recursively subdivided until each *cluster* contains a small number of objects. The binary tree thus obtained now contains separating planes in its interior nodes and lists of objects at its leaves.

Given a viewing position and a separating plane, *no polygon on the viewpoint side can be obstructed by any polygon on the farther side.* A cluster on the viewpoint side will take priority over the cluster on the other side since it is closer to the viewing position. The separating planes determine which side a cluster belongs to. With this, we can now construct a priority list of all the clusters of objects from any viewing position.

The technique of Fuchs and Naylor proceeded to automate the process of choosing the separating planes since Schumacker's method involved hand-picking the planes. This really is imperative for highly dense scenes (with hundreds of thousands of polygons). Their method is, however restricted to polygonal scenes since the separating planes are the planes of the polygons that determine the scene.

One of the polygons was picked as the separating plane. The remaining polygons were divided up into two sets, based on the side of the plane they belong to. Polygons which

crossed this plane were split and each piece went into its appropriate side. Thus the environment need not be linearly separable. As in the earlier algorithm, the two subsets are recursively subdivided picking at each step one of the polygons as the separating plane. The process ends when there are no more polygons left. The Binary Space Partitioned (BSP) tree thus built now contains a polygon at each node with the left and right subtrees containing the polygons on either side of the plane of the polygon.

Traversing the BSP tree is straightforward: paint the farther side first, then the root polygon (which is contained by the separating plane) and lastly the viewpoint side – *a back to front* painting of the polygons.

3 K-D Trees in Ray Tracing

To use such a binary structure for ray tracing the following must be kept in mind:

- We would like to be able to ray trace any kind of object. Therefore, using the planes of polygons as separating planes is restrictive; at the same time the plane must be easy to intersect against so as not to be a bottleneck during traversal.
- Splitting objects across planes is inappropriate for ray tracing and must be completely avoided.
- The generated tree must be well balanced; this can lead to a faster intersection process.
- Use of hierarchical bounding volumes is essential; again, these volumes must be simple enough so as to serve as a quick intersection check.

We propose using planes orthogonal to the X, Y and Z axes as separating planes. It is easy to intersect a ray against any of these planes. The construction of the tree is also

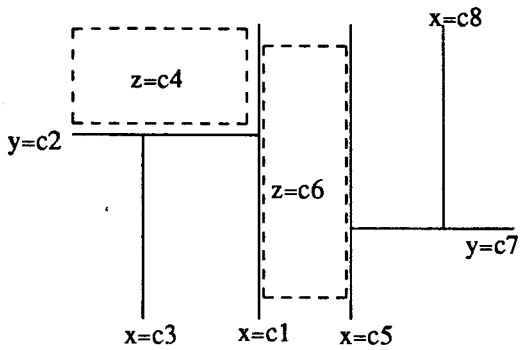

Figure 1: A Sample *k-d* Tree

faster since only comparisons are involved. A sample *k-d* tree is shown in Fig. 1. The dotted lines indicate planes of constant Z (parallel to the plane of the page). Only the separating planes are indicated.

We are going to use the *k-d* tree to search for the closest intersection; splitting objects across planes only serves to increase the size of the data set, which will in turn lead to a slower intersection search. So objects are permitted to be *shared* between regions and we make sure that each object is processed only once however many regions it is a part of.

A well balanced *k-d* tree is required since this means a smaller number of levels in the tree; this will yield a saving during traversal as the search for intersection will in general involve going down a smaller number of levels. In addition, minimizing the shared object count will result in a better subdivision.

Our method uses cuboidal bounding volumes. Intersecting a cuboidal volume is simple. Using tighter volumes [14] is a possibility but the volume intersection times increase and may not be economical for objects like polygons with a small number of edges.

4 Construction of the K-D Tree

Our aim is to partition the object space into two subspaces at each step using planes of the form $x = c_1$, $y = c_2$ or $z = c_3$ which best satisfy our requirements. Two considerations are important for this:

- The number of objects in the two subspaces must be nearly equal.
- The number of objects common to both regions must be kept to a minimum as this leads to a better partition.

The plane best satisfying these requirements is chosen as the separating plane. This process is then recursively applied until the number of objects in each subspace is below a threshold or no suitable plane can be found.

To start this process, we read in all the objects from the database and apply all modeling transforms to position the objects in world space. All objects will now be subjected to a transformation (given the viewing specifications) to eye space [5]. The bounding boxes of all the objects are determined [14], and from this the bounding box for the whole scene is found.

4.1 Choosing a Separating Plane

This process essentially involves a binary search along the three coordinate axes within the scene bounds. Algorithm 1 illustrates the procedure. It consists of the following steps:

1. Plane Choice:

Choose the plane midway between the scene maximum and minimum along the particular coordinate axis.

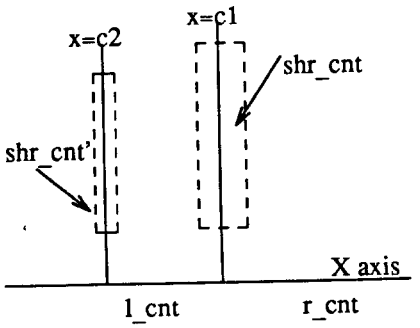

Figure 2: Object Classification

2. Classification:

Classify each object of the scene according to its position on the left, right or both sides of the chosen plane. To determine how good this partition is, compute a figure of merit (*fom*) as follows.

$$fom = absolute(l\_cnt - r\_cnt) + shr\_cnt$$

where *l_cnt, r_cnt, shr_cnt* represent object counts on the left, right and both sides of the plane.

If this is unsatisfactory (determined by the termination conditions), we will need to move the plane to the region that contains the larger number of objects. Whichever region is chosen, *only the objects in that region need to be partitioned* for the new plane choice. To make this clearer, consider the following example.

Let *old_left_cnt* and *old_right_cnt* indicate counts of objects that have been processed. Before starting a plane search along a coordinate direction, say the X axis, these will be initialized to zero. Let $x = c_1$ be the first plane choice resulting in *l_cnt, r_cnt and shr_cnt* (refer to Fig. 2). If the left side contains more objects ($l\_cnt > r\_cnt$) the next choice $x = c_2$ will be to the left of $x = c_1$. Since $c_2 < c_1$ objects to the right of $x = c_1$ will also be to the right of $x = c_2$ and hence need not be processed again. All that needs to be done is

to add *r_cnt* to *old_right_cnt*. We now partition the region $x < c_1$ with respect to the plane $x = c_2$ and if this results in l_cnt', r_cnt', shr_cnt', the actual counts taking all objects into account will be

$$\begin{aligned} l\_cnt'' &= l\_cnt' \\ r\_cnt'' &= r\_cnt' + old\_right\_cnt \end{aligned}$$

and the new figure of merit will be

$$fom' = abs(l\_cnt'' - r\_cnt'') + shr\_cnt'$$

A similar procedure would be adopted if the new plane choice had been to the the right of $x = c_1$. In this case the processed objects will be counted into *old_left_cnt*.

Thus each time we move a plane along a coordinate direction, one of the 'old' counts will be updated.

3. Move left/right:

To choose the new plane as in step 1, we need to update the left and right bounds (in the case of an x plane). If the new plane is to be located to the left of the present plane, then the latter becomes the right bound; if it is to the right, then the newly chosen plane chosen becomes the new left bound.

4. Termination:

If the figure of merit computed above is less than a threshold, then we are done. Else we must determine whether we still need to continue the binary search in this coordinate direction. As illustrated in Algorithm 1, there are three conditions:

- The left and right bounds differ by less than a chosen resolution.

- All objects are shared between the two regions(i.e. no suitable plane can be found for subdivision). In this case the figure of merit will equal the original number of objects.
- The number of objects in both subspaces are equal (in this case the search proceeds to the next dimension).

4.2 Building the K-D tree

Knowing how to determine a separating plane building the k-d tree is straightforward. Algorithm 2 illustrates the procedure. Given as input the bounds of the scene in the three directions, the list of input objects and their number, we recursively subdivide the object set. The process terminates when an object set contains less than a small number of objects (which can be set in advance) or if no plane can be determined to subdivide the object list. In these cases the object list is stored at a leaf node in the tree.

5 Determining the Closest Intersection

Let a ray be represented by the following parametric equation(Fig. 3):

$$\begin{aligned} \vec{r}(t) &= \vec{p} + t\vec{d} \\ with & \\ \vec{r}(0) &= \vec{p} \\ \vec{r}(1) &= \vec{p} + \vec{d} = \vec{q} \end{aligned}$$

As a ray preprocess, $\vec{q}$ must be determined such that it is beyond the extent of the scene. During traversal, only intersections between ray end points are counted. For first generation rays, this can be ensured by placing the projection plane behind the scene. For

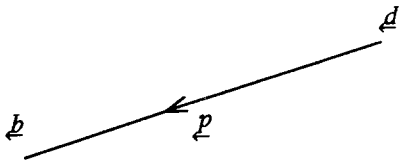

Figure 3: Parametric definition of a ray

higher generation rays, the ray is made at least as long as the largest distance within the scene extent.

5.1 The Traversal Algorithm

Given a ray segment and the root of the *k-d* tree we see whether the root is a tree node or a leaf node. If it is a leaf node, then all the objects in its list will be intersected, and the closest intersection (if any) will be reported. Of course, we use bounding boxes for the objects, and only if the bounding box is intersected do we need to intersect the object within it. On the other hand, if this is a tree node, we have two different cases:

- The ray segment lies entirely on one side of the separating plane.
- The ray segment lies on both sides of the plane(i.e., the ray crosses the plane).

These two cases are easily determined by looking at the appropriate coordinates of the ray end-points based on the orientation of the separating plane.

For case 1 we need to search the subspace containing the ray segment. For case 2, we intersect the ray with the separating plane and split the ray into two parts. We have to examine both subspaces, but we always start with the subspace that is closer to the ray origin (and the eye). If we do find an intersection internal to this subspace, then we are done, else we will need to examine the second region. This process is recursively applied until we find an intersection or run out of regions (and hence objects).

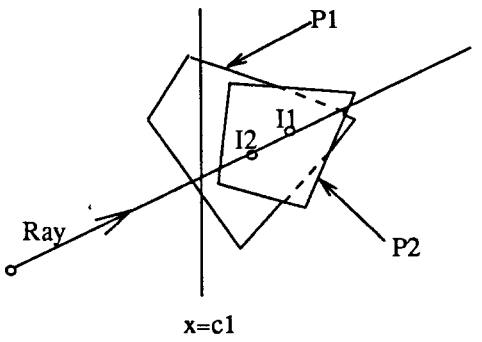

Figure 4: A Case 2 Possibility

Case 2 has a complicating factor. An intersection from the first of the two regions could be erroneous, as the intersected object could be part of several subspaces. Before claiming this as the closest intersection, we must make sure that the t value at this intersection is within the ray segment for this subspace. We can do this since we know the t values at the ray segment end points. Fig. 4 illustrates such a case with a separating plane $y = c_1$.

P_1 and P_2 are two polygons. P_1 is part of both regions whereas P_2 is entirely to the right of the separating plane $x = c_1$. Although P_1 seems to be closer to the ray origin, the closer intersection is at I_2 with polygon P_2. In this case the x coordinate of I_1 will be greater than c_1 necessitating the intersection check with P_2.

A bit flag must be maintained with each object to indicate whether or not it has been processed during the intersection search. Only objects which have not been processed will then be examined. This also means that all the flags that were set must be reset before we do an intersection search for the next ray.

6 Test Images

This algorithm has been coded in C on a VAX 11/780 running 4.3 Berkeley UNIX[1]. We actually tested our program on a Sequent Balance 2000 due to time constraints on the 780. A Sequent NEC processor benchmarks at about 0.5 times a VAX 780 processor (Courtesy Roy Jenevein, Dept. of Computer Sciences, The University of Texas at Austin). These normalized timings appear in Table 1 which describes the statistics of our test cases. In this table, 'Avg. obj. at any leaf' indicates the average number of objects at any leaf node of the *k-d* tree while 'Max. obj. at any leaf' indicates the maximum number of objects at any leaf node. The rest of the table is self-explanatory. All images were computed at a resolution of 512×512. For antialiasing, each pixel was supersampled on a 3×3 grid and averaged.

Fig. 5 is the familiar 'recursive pyramid' of Andrew Glassner and Fig. 6 is the Caltech tree (distributed by Timothy L. Kay, California Institute of Technology). These were reproduced to compare our technique with those of other researchers in this area [1][9][14].

Fig. 7 is the Utah teapot (designed by Martin Newell) resting on a base polygon. This image is a combination of Bezier patches (handle and spout) and surfaces of revolution (body). The Bezier patches were subdivided [15] into triangles and normal interpolation was used for smoothing. The profile curves of the surfaces of revolution were subdivided until they could be approximated by straight line segments. These were then rendered as truncated cone frustums.

Fig. 8 illustrates the structure of a DNA molecule (Courtesy of Professor Langridge and Dr. Pattabiraman, School of Pharmacy, University of California, San Francisco). This image consists of 410 spheres and represents one turn of a double helix.

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

Fig. 9 consists of three B-Spline patches subdivided into 4818 triangles (Courtesy of Gordon Fossum, University of Texas at Austin). Normal interpolation was used for smoothing.

Fig. 10 and Fig. 11 are test databases distributed by Eric Haines [10]. These figures should be helpful to other researchers for performance comparisons.

7 Results

The 'recursive pyramid' and the 'tree' are the only scenes with which we are able to compare our method with those of other researchers. With our technique the pyramid runs significantly faster than the methods of Kay & Kajiya or Glassner but a little slower than James Arvo's method. The reason is clear. The total number of object intersections makes the difference. Also note that our technique uses fewer bounding volume intersections than that of Kay & Kajiya although they were using a much tighter bounding volume. This clearly demonstrates the power of this technique to avoid most of unnecessary bounding volume intersections.

It is somewhat different with the tree image. Here our technique runs a little faster than that of Arvo, whose timing was previously the best. In this case the number of object intersections makes the difference. Our technique interrogates only a little over half the number of objects that Arvo's '5D Space' technique does. However this advantage is somewhat offset by the bounding volume intersections that need to be done during the traversal.

The pyramid takes longer to compute because in this image the majority of the rays go through the scene without hitting anything. Arvo's method will produce beams with null candidate sets so that such rays are trivial to deal with. Caching such rays will make the intersection search even faster. In our technique, however, the overhead of bounding

box intersections will slow down the intersection search. This also explains the fact that on the tree image which contains a much higher percentage of hitting rays, our method gains an advantage.

The last two images take considerably longer times to compute because of the large number of spawned rays. Also, the percentage of background in these scenes is significantly lower than the other scenes. The average number of objects interrogated by each ray in these scenes is very small.

8 Limitations

Our method is by no means without its limitations. The fact that our separating planes have to be oriented orthogonal to the principal axes could lead to bad cases where it would be impossible to find a separating plane for a large number of objects. Taking a look at the statistics of the arches image, we find that the maximum number of objects at any leaf node is 33. In this case all 33 triangles were oriented in such a way that their extent projections onto the three coordinate axes all overlap. Hence there is no separating plane along any of the three coordinate axes that would put at least one object on either side. Thus if this leaf list has to be examined by the ray tracer for an intersection, then all objects have to be tested.

Of course, for polygonally faceted objects the solution to this problem is to use the standard BSP tree [6][7][16] proposed by Fuchs, Naylor et al. However if we have non-polygonal objects, then finding a suitable plane is a much harder problem. It must be remembered that using arbitrarily oriented planes will in general increase the traversal cost since they require more computation for intersection than axis-aligned planes. This consideration may not be relevant for machines which have special-purpose hardware for performing these intersection calculations, in which case a BSP-tree version of our algorithm would

likely be superior.

9 Future Work

So far, our goal has been to build a balanced *k-d* tree with respect to the number of objects. This inherently assumes that objects are of similar size. In general, the size of objects has to be taken into account when we compute the Figure of Merit (*fom*). As our bounding volumes should be as tight as possible, we are interested in partitioning the object set so that the sum of the bounding volumes is a minimum. A good example is the 'Tree' image. The tree rests on a large base polygon. If we just subdivide based on the number of objects (as is done now) on either side of the partition, then the base polygon will make all the volumes in the hierarchy have large void spaces. In this case, the first plane has to be parallel to the base polygon thereby separating it out from the rest of the image. Since our current implementation makes no attempt to minimize bounding volumes, we had to hand-pick the first separating plane.

10 Conclusion

We have presented an efficient method of ray tracing using *k-d* trees. Algorithms to build and traverse this data structure have been given. We have demonstrated that this technique is efficient in handling complex scenes, and that it is among the fastest known ray-tracing methods.

We feel that no single algorithm can be optimal on every kind of scene as each algorithm uses scene coherence in a different way. Our present efforts are in trying to get at a better understanding of this so that we may be able to use appropriate techniques, depending on the properties of the scene we are trying to render.

11 Acknowledgements

We would like to thank our colleagues A. T. Campbell and Gordon Fossum for their invaluable suggestions and constructive criticism. Our acknowledgements to Tim Kay and Eric Haines for making available their data bases to us. Special thanks to Nick Shinichiro Haruyama for his display software.

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. Data structures for range searching. *Computing Surveys*, 11(4). December 1979.

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

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

[5] James D. Foley and Van Dam A. *Fundamentals of Interactive Computer Graphics.* Addison Wesley, Reading, Massachusetts, 1982.

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

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

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

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

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

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

[15] Jeffrey M. Lane, Loren C. Carpenter, Turner Whitted, and James F. Blinn. Scan line methods for displaying parametrically defined surfaces. *Communications of the ACM*, 23(1), January 1980.

[16] Bruce F. Naylor. *A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes.* PhD thesis, The University of Texas at Dallas, May 1981.

[17] William M. Newman and Robert F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, 1979.

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

[19] Robert A. Schumacker, B. Brand, M. Gilliland, and W. Sharp. *Study for Applying Computer-Generated Images to Visual Simulation.* Technical Report AFHRL-TR-69-14, U. S. Air Force Human Resources Laboratory, September 1969.

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

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

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

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

Ray and Intersection counts are in thousands

Image	Pyramid	Tree Branches	Tree Leaves	DNA	Teapot	Arches
Objects	1024	1270	7454	410	4513	4818
Light Sources	1	1	1	1	2	2
Total rays	299	395	412	356	499	437
Visual rays	262	262	262	262	262	317
Shadow rays	37	133	150	94	237	120
Rays that hit	44	149	193	98	144	73
Bound. Vol. Inters.	2040	4282	4983	3248	7278	6756
Obj. Inters.	251	325	441	680	1883	959
Obj. Inters./ray hit	3.2	1.3	1.8	4.0	6.5	5.7
Avg. obj. at any leaf	3.4	2.9	3.5	6.6	5	8.0
Max. obj. at any leaf	4	6	10	11	12	33
Time (VAX 780-Hr:min)	0:17	0:33	0:47	0:31	1:23	0:54
Time/ray (sec)	0.0036	0.0048	0.0067	0.008	0.01	0.008
Time/ray hit (sec)	0.025	0.014	0.017	0.02	0.036	0.048
k-d Tree build (min)	0.4	0.7	8.0	1.2	4.1	5.2

Image	Sphere Flake	Cornell Tree
Objects	7382	8191
Light Sources	3	7
Total rays	1819	1677
Visual rays	722	453
Shadow rays	1097	1224
Rays that hit	873	242
Bound. Vol. Inters.	20484	11859
Obj. Inters.	2454	808
Obj. Inters./ray hit	2.052	1.5
Avg. obj. at any leaf	2.8	3.0
Max. obj. at any leaf	5	5
Time (Sun 3/50-Hr:min)	4:17	2:12
Time/ray (sec)	0.0085	0.0047
Time/ray hit (sec)	0.0177	0.032
k-d Tree build (min)	3.6	3.6

Table 1. Statistics of Test Images.

Input: list of objects, number of objects, scene volume.
Output: separating plane, divisible flag.

```
choose_plane (object_list, object_cnt, vol, plane, divisible)
{
  i = X
  fom = best_fom = -∞
  while (fom > threshold && i ∈ (X,Y,Z))
  {
      choice = (vol_{i_min} + vol_{i_max})/2
      classify (obj_list, choice, &new_side, &fom)
      if (new_side == LEFT)
        vol_{i_max} = choice
      else (if new_side == RIGHT)
        vol_{i_min} = choice
      else i++
      if (fom < best_fom)
      {
        best_fom = fom
        plane→value = choice
        plane→indx = i
      }
      if (|vol_{i_min} − vol_{i_max}| < Resolution_i)
        or (fom == object_cnt)
        i++
  }
  *divisible = (fom == object_cnt)
}
```

Algorithm 1. Choosing a separating plane.

Input: object list, object_cnt, scene volume.
Output: root of the k-d tree.

```
TREE_PTR build_k-d (object_list, object_cnt, volume)
{
  choose_plane (obj_list, object_cnt, volume, &plane, &divisible)
  if (divisible)
  {
    create "root node"
    classify (obj_list, plane, &l_list,&r_list &l_cnt, &r_cnt)
    if (left_cnt > threshold)
      root→left = build_k-d (l_list, l_cnt, getvol(l_list))
    if (root→left == NULL)
      root→left = obj_list
    if (right_cnt > threshold)
      root→right = build_k-d (r_list, r_cnt, getvol(r_list))
    if (root→right == NULL)
      root→right = obj_list
    return (root)
  }
  return (NULL)
}
```

Algorithm 2. Building the *k-d* tree.

Input: root of the k-d tree, ray end points.
Output: true/false from the function and if true the intersection.

```
traverse (root,ray,tstart,tend,inters)
{
  if (root→type == LEAF)
    return (list_inters (root→list,tstart,tend,inters))
  else
  {
    path = ray_path (root,ray,tstart,tend,&tmid)
    switch (path)
    {
       LR : if (traverse (root→left,ray,tstart,tmid,inters))
              return (TRUE)
            return (traverse (root→right,ray,tmid,tend,
                                          inters))
       RL : if (traverse (root→right,ray,tmid,tend, inters))
              return (TRUE)
            return (traverse (root→left,ray,tmid,tend,
                                          inters))
       L  : return (traverse (root→left,ray,tstart,tend,
                                          inters))
       R  : return (traverse (root→right,ray,tstart,tend,
                                          inters))
    }
  }
}
```

Algorithm 3. Traversing the *k-d* tree.