The Visual Computer

Heuristics for ray tracing using space subdivision

J. David MacDonald[1] and Kellogg S. Booth[2]

[1] Visual Edge Software Ltd., Montreal, Quebec, Canada
[2] Computer Graphics Laboratory, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

Ray tracing requires testing of many rays to determine intersections with objects. A way of reducing the computation is to organize objects into hierarchical data structures. We examine two heuristics for space subdivisions using bintrees, one based on the intuition that surface area is a good estimate of intersection probability, one based on the fact that the optimal splitting plane lies between the spatial median and the object median planes of a volume. Traversal algorithms using cross links between nodes are presented as generalizations of ropes in octrees. Simulations of the surface area heuristic and the cross link scheme are presented. These results generalize to other hierarchical data structures.

Key words: Octree – Ray tracing – Space subdivision – Splitting plane – Surface area

The Visual Computer (1990) 6:153–166

1 Introduction

Ray tracing is a popular algorithm for computer rendering of synthetic images (Glassner 1987a). The main reason why the use of ray tracing is so widespread is its simplicity of coding and the comparative ease with which ray tracing renders many realistic effects including shadows, penumbrae, reflection, refraction (transparency), and motion blur (Cook et al. 1984). The principal drawback of ray tracing is its comparatively high computational cost, which is due primarily to the high occurrence of one basic operation, the *ray-scene intersection* test. The simplest, brute-force method of determining the ray-scene intersection is to test the ray against each object, remembering which object, if any, has the nearest point of intersection. This has been vastly improved with the use of *scene structuring* (Fujimoto et al. 1986; Glassner 1984, 1987b, 1988; Goldsmith and Salmon 1987; Kaplan 1985; Kay and Kajiya 1986; Scherson and Caspary 1987; Rubin and Whitted 1980; Weghorst et al. 1984), which reduces the number of *ray-object intersection* tests required.

Scenes are modeled with a variety of different implicitly and explicitly defined objects and surfaces. They range from simple objects, such as spheres, ellipses, triangles, polygons, and parallelepipeds, to more complex surfaces such as cubic patches, spline surfaces, and implicit functions. For all but the simplest of these, an intersection test of a ray with the object is a nontrivial computation. To speed up the intersection test, a *bounding volume* is placed around the object. The bounding volume is typically a very simple type of object with an easy intersection test, such as a sphere or a parallelepiped with sides perpendicular to the major axes. In order to determine whether a ray intersects a particular object, the ray is first tested against the object's bounding volume. If the ray does not intersect the bounding volume, it does not intersect the object inside. Otherwise, the ray must be tested against the object in the usual manner. A common type of object for bounding volumes is a rectangular parallelepiped or *box* with each side perpendicular to a major axis.

The notion of a bounding box generalizes to the idea of scene structuring with a *hierarchical data structure.* There are two main classes of hierarchy applicable to ordering the scene, one a dual of the other. *Object subdivision* clusters the objects composing a scene, recording the space that each object inhabits. *Space subdivision* subdivides space, recording the objects that inhabit each region of space.

A *hierarchical extent tree* is a recursive subdivision of objects. The root of the tree corresponds to a bounding volume containing all of the objects in the scene. The children of a node correspond to a set of bounding volumes that divide the objects contained in the node's bounding volume. When the number of objects in a node's bounding volume is one, the node is given a single child where the object is actually stored. Although reference is made to objects enclosed by, or contained within, a node's bounding volume, it should be observed that objects are actually only stored in the leaves. A number of algorithms to build object subdivisions have been reported (Goldsmith and Salmon 1987; Kingdon 1986).

The dual of object subdivision is space subdivision, which subdivides space into disjoint subregions, recording the objects that inhabit each subset of space. The *octree* is a common type of space subdivision. Initially, the octree consists of only one node, representing the bounding volume containing all of the objects in the scene, exactly the same as the root of a hierarchical extent tree. Using three *splitting planes*, one perpendicular to each of the three major axes, the bounding volume is divided into eight smaller volumes, each of these eight a child of the root (hence the term "octree"). Every object is placed in whichever child encloses it. Each of the children may be recursively subdivided.

The bounding volumes associated with nodes are usually referred to as *voxels*, which is the three-dimensional analog of a pixel. Sometimes an object belongs in more than one voxel. In this case, either the object is split into new objects that do not belong in more than one node's voxel, or the object (more often a pointer to the object) is stored in both nodes (Fujimoto et al. 1986; Glassner 1984; Kaplan 1985). As with the hierarchical extent tree, the resulting octree has all of its objects stored at the leaves and none in the interior nodes. Unlike the hierarchical extent tree, a single leaf may contain more than one object.

If a ray intersects the root node of an octree, it is recursively tested against the children of the intersected node. When a leaf node is intersected, all of the objects stored in it are tested for intersection and the nearest, if any, is recorded. The octree allows testing nodes in the order that the ray passes through them, because it subdivides space into disjoint regions. For this reason, the traversal algorithm can halt as soon as it finds a leaf in which an object is intersected.

The splitting plane for each axis of subdivision in an octree may be any arbitrary plane within the current volume. Often the plane that is halfway between the limits of the volume, the *spatial median*, is chosen. We refer to this as *uniform space subdivision*. Choosing the spatial median means that the positions of the planes need not be stored in each node because they can be generated from knowledge of the limits of the node. Depending on the traversal method, the storage saved may be large enough to warrant the additional re-computation of the spatial median during traversal.

The two-way analog of the eight-way octree is the *k-d tree* or *bintree* (Samet 1984). The only difference is that where the octree divides a node into eight subnodes using three splitting planes, a bintree divides a node into only two subnodes using just one splitting plane. Any octree can be represented by a corresponding bintree. The subdivision of a node in an octree is represented by three levels of subdivision of a node in a bintree. Not all bintree subdivisions can be represented exactly by an octree. It is often more convenient and more efficient to use bintrees for space subdivision (Kaplan 1985).

There is an important clarification to be made concerning the determination of whether a certain object belongs in a given node of an octree (or a bintree). An object belongs in a node only if the surface of the object intersects the node's box. The reason for this is that the point of intersection of a ray with an object cannot occur within a box that does not contain some part of the surface.

Octrees and bintrees share a problem peculiar to space subdivision hierarchies. Depending on the implementation, an object may be stored in more than one node and may not be totally enclosed by any particular node. Therefore, an intersection test of a ray with an object may find an intersection point outside the volume of the current node. This is called *fragmentation.*

The algorithm as described so far assumes that the computed intersection is the nearest point of intersection and halts. However, the intersection point may be outside the volume of the current node, so we have no guarantee that there is not a closer intersection point with some other object in the scene that is in some other node. Because of this, only ray-object intersections that occur within the volume corresponding to the current node are valid and other intersections must be ignored until the appropriate node is examined. To avoid testing a ray with the same object more than one time, a *ray-object cache* can be used. This technique was suggested by Amanatides and Woo (1987) and independently by Arnaldi et al. (1987), who used the term "mailbox" to describe the cache. Caching has been incorporated into some ray tracing algorithms that use space subdivision (Cleary and Wyvill 1988).

To implement the ray-object intersection cache, it is sufficient to maintain for each object the identity of the most recent ray that has been found to intersect the object and the point at which the intersection occurs. Subsequent intersection tests simply check to see if the object has been tested against the current ray and reuse the intersection data if it is available. For uniprocessors it might be reasonable to cache the intersection information with each object simply as an additional field stored with the object description. For a multiprocessor in which every ray is assigned to a different processor, however, it would be necessary to have a separate cache for every ray or processor, because multiple rays might be tested in parallel against a single object. A single cache keeping only the "most recent" intersection for an object might throw away information about an active ray if a second ray hit the same object. Adding enough fields to each object to cache intersections from every processor could prove costly in storage, so a hashing scheme or similar technique might be required to maintain the cache. We assume that some type of cache is used for all subdivision algorithms.

In the following sections we review three particular space subdivision techniques in terms of their costs for *construction*, *traversal*, and *storage*. We then introduce two heuristics for constructing space subdivisions and a neighbor link strategy for improving traversal and storage costs. We report on simulations that test these ideas using bintree implementations.

2 Previous space subdivision algorithms

Glassner gives one of the earliest published applications of octrees to ray tracing using the spatial median splitting planes (Glassner 1984), with later papers elaborating on the technique (Glassner 1987b, 1988). Glassner's method of construction is a simple breadth-first technique. Nodes which have more than a certain number of objects are subdivided until a predetermined size of tree is reached. The tree building is governed by two parameters: the maximum number of nodes and the threshold value used for determining whether to split a node. In some cases, Glassner's algorithm will subdivide a smaller volume and leave a larger volume unsubdivided. It is likely that only a few rays go through the small volume, while many intersect the large volume. Therefore, subdividing the smaller gives very little performance gain. It is probably better to subdivide the larger.

The crux of the problem is that Glassner's algorithm does not take into account any measure of the chance of a ray intersecting a node. Glassner presents an improved algorithm (Glassner 1987b) in which a node is subdivided if it contains more than a threshold number of objects, or if it is larger than a given volume. It seems that the choice of threshold is very critical to the performance of this algorithm.

During ray tracing, the ray progresses through the volumes defined by the leaves of the octree, enumerating the leaf nodes intersected by the ray in order of nearness to the ray origin. The objects within the enumerated leaves are tested for intersection and the ray tracing algorithm halts at the first intersected object. Each time the ray searches for a new leaf of the octree, the traversal procedure starts at the root node and works down the tree node by node until a leaf is found. But two consecutive leaves along the path of a ray generally share several ancestor nodes. Glassner's approach ignores this. A simple optimization of Glassner's traversal algorithm would be to perform a binary search among the ancestors for the lowest common ancestor. Even with this optimization, we suspect that for really large octrees the double-logarithmic search time would still be a significant overhead. Perhaps the worst drawback to Glassner's traversal algorithm is the problem of ensuring that a "good" hash function exists, since this is the mechanism used for rapid accessing of nodes in the octree. This is not adequately described by Glassner for large octrees. A basic problem seems to be that Glassner's approach is geometric in nature and ignores the connectivity (or topology) implicit in the octree.

Kaplan (1985) describes an implementation of a bintree very similar to Glassner's octree approach. A node is subdivided at the spatial median in each of the three coordinates and three levels of subnodes are created to represent the subdivision. The traversal algorithm for a bintree is simpler because only a two-way decision is required at each node, instead of an eight-way decision required for each octree node. The bintree representation typically results in fewer leaves than the corresponding octree, because each leaf in a bintree corresponds to at least one and possibly as many as four leaves in the corresponding octree. The construction of the tree is governed by the same criteria as Glassner's second method (Glassner 1987b). A node is subdivided if it contains more than a threshold number of objects, or if it is larger than a threshold size. Kaplan suggests using one as the threshold number of objects. The problems with this approach are the same as for Glassner's method.

Fujimoto et al. (1986) described what they consider to be a significant speed breakthrough with regard to space subdivision structures for ray tracing. Their ARTS (accelerated ray tracing system) implementation is distinguished from Glassner's method by the speed of its traversal algorithm, as opposed to the uniqueness of its octree. The traversal algorithm uses incremental integer arithmetic similar to Bresenham's algorithm to enumerate the space through which a ray travels. This is a three-dimensional adaptation of the standard two-dimensional DDA (digital differential analyzer) used to draw lines. ARTS uses a uniform space subdivision with explicit storage of the octree as a tree. This method is superior to Glassner's hash table strategy in terms of storage, requiring about 16% less space according to the published storage requirements for both algorithms (MacDonald 1988).

In addition to being more compact, the ARTS method has faster traversal times because of the explicit links to the children and because space is partitioned into small voxels of a fixed size. The smallest leaf in the octree is a power of two times the size of the underlying voxels. The splitting planes of the octree coincide with faces of the underlying voxels, allowing a straightforward mapping from an underlying voxel to a leaf node. The ARTS system traverses upwards in the octree from the previous leaf only as far as required and then down to the adjacent leaf. It is claimed that this can be done quite efficiently using byproducts of the incremental integer arithmetic algorithm.

We see three basic bottlenecks in the published descriptions of these space subdivision algorithms: the construction of optimal hierarchies given a fixed number of nodes, the traversal time as rays are traced through volumes, and the storage costs associated with individual nodes. These issues are addressed in turn in the following sections.

3 The surface area heuristic

The construction of the bintree or octree is typically insignificant compared to the computation spent in actually traversing the tree to determine ray-object intersections. Therefore it would be advantageous to devote a greater effort to creating a more efficient tree, under the assumption that the extra time would then be recovered during tree traversal.

A heuristic approach for bintree construction can be derived from the observation that the number of rays likely to intersect a convex object is roughly proportional to its surface area, assuming that the ray origins and directions are uniformly distributed throughout object space and that all origins are sufficiently far from the object (Stone 1975). This heuristic has been used to provide a measure of the likelihood that a ray will intersect a bounding volume in a hierarchical extent tree (Goldsmith and Salmon 1987) and in octrees (Cleary and Wyvill 1988). We derive similar predictions for the number of objects, interior nodes, and leaves intersected in a space subdivision hierarchy and use these to govern the construction of the tree.

We assume that all rays intersect the bounding volume for the entire scene. Thus, every ray intersects the root voxel. We further assume that the probability of a ray intersecting any interior or exterior node is equal to the surface area of the node divided by the surface area of the root. This results in the following intersection estimates.

no. of interior nodes hit per ray

$$= \sum_{i=1}^{N_i} SA(i)/SA(root)$$

no. of leaves hit per ray

$$= \sum_{l=1}^{N_l} SA(l)/SA(root)$$

no. of objects tested for intersection per ray

$$= \sum_{l=1}^{N_l} SA(l) \cdot N(l)/SA(root)$$

where the various quantities are

N_i = no. of interior nodes

N_l = no. of leaves

$N(l)$ = no. of objects stored in leaf l

$SA(i)$ = surface area of interior node i

$SA(l)$ = surface area of leaf node l

Given these measures of the node, leaf, and object visists performed during traversal of the tree, an estimate of the cost of the tree can be obtained. The costs associated with these three components depend on the particular implementation of the traversal algorithm and may be determined theoretically or experimentally. The total cost of a particular tree is determined from the three sums above and the three related costs, which are assumed to be constants for a given implementation.

This is expressed as cost of tree

$$= \frac{C_i \cdot \sum_{i=1}^{N_i} SA(i) + C_l \cdot \sum_{l=1}^{N_l} SA(l) + C_o \cdot \sum_{l=1}^{N_l} SA(l) \cdot N(l)}{SA(root)}$$

where the new quantities introduced in the equation are

C_i = cost of traversing an interior node

C_l = cost of traversing a leaf

C_o = cost of testing an object for intersection

This cost function assumes that rays do not intersect any objects, but also represents an upper bound for rays that do intersect objects. The cost function implies that if an object occurs in two or more leaves, it is tested for intersection each time a ray intersects one of these leaves. Therefore a given object may be tested against the same ray several times. As observed before, this is usually unacceptable, and is avoided by caching objects intersected against a ray so that each object is tested at most once per ray. The cost function given above must be modified to account for this caching based on assumptions about the scene.

To derive the correct cost function, we require a measure of the probability that a ray intersects at least one leaf from the set of leaves within which a particular object resides. This is equivalent to determining the probability that a ray intersects the volume defined by the union of the set of leaves. Because this union may be nonconvex, the probability of ray intersection must be estimated by finding a convex region to approximate the nonconvex region. A simple approximation is the sum of the areas of the projection of the set onto the six faces of the root bounding volume divided by the root bounding volume's surface area. For a convex object, this measure is exactly equal to its surface area divided by the root bounding volume's surface area. We can us this approximation for the set of leaves for all objects, whether the set of leaves for each object is convex or not. This makes the object portion of the cost of a tree object cost per ray

$$= \frac{C_o \cdot \sum_{o=1}^{N_o} SA\,set(S_l(o))}{SA(root)}$$

where the new quantities are

N_o = no. of objects

$S_l(o)$ = leaves in which object o resides

$SA\,set(s)$ = approximate surface area of set s

If we assume that the above costs are accurate, we can use these equations to govern the construction of the tree, choosing nodes to subdivide so as to minimize the total cost of the tree for a given number of nodes in the tree. We call this rule the *surface area heuristic*. It generalizes Glassner's use of a minimum size below which nodes are not subdivided.

The validity of the surface area heuristic was tested using a simulation. A set of 100 boxes with random sizes and positions was created, where each box was a standard rectangular parallelepiped and 100000 random rays were traced through the bounding volume enclosing the boxes. These rays had origins outside the bounding volume and were directed at the bounding volume. The statistics recorded are presented in graphical form in Fig. 1, where each point represents the surface area of a box and the number of rays which intersected the box. The number of rays intersecting a box is thus shown to be directly proportional to its surface area to within statistical variation.

The graph in Fig. 1 illustrates that the number of rays intersecting a box is proportional to its surface area, assuming random rays. However, this does not prove that the estimates of interior and leaf nodes intersected are correct, because the search is truncated as soon as an intersection is found. The estimate of the number of object tests also cannot be assumed to be proven because that esti-

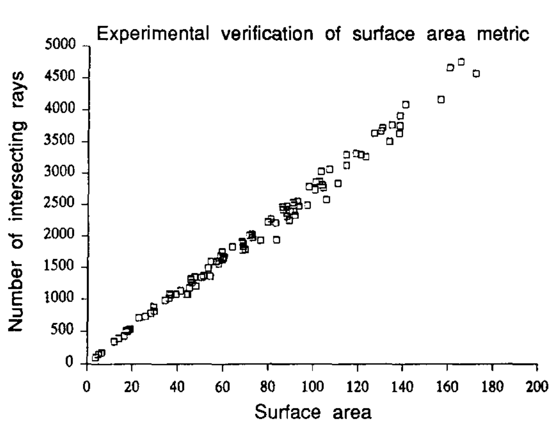

Fig. 1. Surface area heuristic data

mate is derived from an approximation of a possibly concave set of leaves by a convex volume. To test the validity of these estimates, a further simulation was performed.

Random scenes of objects and random bintrees were created. These were used to trace random rays as in the previous simulation. The estimated numbers of interior nodes, leaves, and objects visisted were compared with the actual numbers from the ray tracing. Each scene contained a random number of objects between 10 and 500, with random distribution in size from 0.01 to 1. The bintree created for the scene contained a random number of nodes between 10 and 1000, in which nodes were subdivided in random order along a random axis at a random position within the corresponding voxels. In all 529 random scenes were created and 10000 rays were traced for each scene. Table 1 summarizes the results of the simulation.

In all cases the actual number is proportional to the estimated number. In the case of the number of interior nodes and leaves intersected, the estimates actually provide upper bounds rather than an average case estimate. This is understandable, as the derivation of the estimates assumes that the rays hit no objects. The constants of proportionality may therefore be used in conjunction with the surface area heuristic to give a more accurate estimate of the average number of interior nodes and leaves intersected. The estimate of the number of objects intersected was shown to be quite accurate, with a constant of proportionality close to one.

One reason that this provided an average case estimate, rather than an upper bound, is that there are too few objects in the scene. Truncating the search as soon as an intersection was found probably did not save many intersection tests, because each ray may have intersected zero or one objects. Therefore the estimate provided an average case estimate. With denser scenes, the object intersection estimate should probably be scaled down in the same way as the interior and leaf node estimates.

4 Spatial median versus object median

In all of the octree and bintree constructions the position of the splitting planes is arbitrary, even if the surface area heuristic is employed. Traditionally, the splitting plane is chosen as the spatial median, resulting in a uniform space subdivision. Heckbert (1982) employed a *median split algorithm* that chooses a splitting plane based on the *object median* in a k-d tree, where the objects are color triplets (single points). The object median is the splitting plane that places one half of the objects on each side of the plane. The cost estimate developed using the surface area heuristic can also be applied to selecting "good" splitting planes in this extended model.

In the following discussions of splitting planes, we will only consider the bintree. We assume that only major planes are used as splitting planes and we ignore the possibility of an object straddling a split-

Table 1. Results of simulation

Quantity	Actual	SD	Correlation coefficient
No. of rays intersecting box	27.5 × surface area	5.2%	0.995
No. of interior nodes intersected	0.752 × estimate	12.7%	0.945
No. of leaves intersected	0.831 × estimate	14.1%	0.900
No. of object tests	1.03 × estimate	9.5%	0.985

ting plane (a case of practical importance, but one we ignore nevertheless). We have to choose a parameter b to position the splitting plane, where $b=0$ corresponds to the lower limit of the splitting plane and $b=1$ is the upper limit. Choosing $b=0.5$ is equivalent to selecting the spatial median.

Let us look at the cost as a function of this parameter b. We observe that the internal node and leaf node components of this cost savings function are constant with respect to b. For the purposes of minimizing cost, we can minimize the function

$$f(b)=LSA(b)\cdot L(b)+RSA(b)\cdot(n-L(b))-SA\cdot n$$

where n is the number of objects in the node, $L(b)$ is the number of objects to the left of the plane at b, and $n-L(b)$ is the number to the right of the plane because of our assumption that no objects straddle the plane. The surface area of the left and right subnodes are $LSA(b)$ and $RSA(b)$, respectively, and the surface area of the node itself is SA. The first term represents the probability that a ray intersects the left subnode multiplied by the number of intersection tests performed in the left subnode. The second term is a similar quantity for the right subnode. The $SA\cdot n$ term is the amount of work required if the node were not subdivided and thus is an amount of work saved by changing the original node from a leaf to an internal node, hence the minus sign. This last quantity is a constant with respect to b, so it may be removed from the function, resulting in the following function to be minimized:

$$f(b)=LSA(b)\cdot L(b)+RSA(b)\cdot(n-L(b))$$

To find a "good" splitting plane, one might evaluate this function at several different positions and choose the position with the minimum value. However, let us examine the behavior of this function. The value of this function at the spatial median is

$$f(0.5)=n\cdot LSA(0.5)$$

because $LSA(0.5)=RSA(0.5)$. Curiously enough, the value of this function at the object median, where half of the objects are on each side of the splitting plane and $L(b)=\frac{n}{2}$, is exactly the same

$$(LSA(b)+RSA(b))\cdot\frac{n}{2}=n\cdot LSA(0.5)$$

because $LSA(b)+RSA(b)$ is a constant independent of b, which means that we can substitute $LSA(0.5)+RSA(0.5)$, which is $2\cdot LSA(0.5)$. This shows that picking the object median results in the same gain as picking the spatial median. Intuitively, one might assume that picking the object median would be a reasonable heuristic for choosing an arbitrary splitting plane, but the above observation indicates that it is equivalent to the standard spatial median subdivision.

The optimum heuristic is to pick the splitting plane which minimizes $f(b)$. Differentiating with respect to b gives

$$f'(b)=LSA'(b)\cdot L(b)+LSA(b)\cdot L'(b)+n\cdot RSA'(b)$$
$$-RSA'(b)\cdot L(b)-RSA(b)\cdot L'(b)$$

which can be simplified by substituting $-LSA'(b)$ for $RSA'(b)$ because $LSA(b)+RSA(b)$ is a constant, giving

$$f'(b)=(2\cdot L(b)-n)\cdot LSA'(b)+(LSA(b)$$
$$-RSA(b))\cdot L'(b).$$

Since $L(b)$ is a discontinuous function, $L'(b)$ is not defined. However, for the purposes of minimization of $f(b)$, we can assume that $L'(b)$ is always nonnegative (the number of objects stored in the left subnode cannot decrease as b increases).

Let us investigate the case where the object median lies at some point $b<0.5$. To the left of the object median, $f'(b)$ is negative, because $L(b)<\frac{n}{2}$ and $LSA(b)<RSA(b)$. To the right of the spatial median, $f'(b)$ is positive, because $L(b)<\frac{n}{2}$ and $LSA(b)>RSA(b)$. Therefore the minimum must occur between the object median and the spatial median in the case where the object median is to the left of the spatial median. A similar argument can be used for the other case where the object median is to the right of the spatial median, thereby proving that for any node and set of objects within it, the optimum splitting plane occurs between the object median and the spatial median, reducing the required search range.

The optimum splitting plane actually occurs within this reduced range and at the upper or lower edge of one of the objects within the range, rather than in the middle of "white space." To take advantage of this reduced range, one must first find the object median, which is easy if the objects are sorted, but otherwise requires a search of the space. If one does not want to perform this search, one can determine how many objects are on each side of the spatial median, thereby determining on which side of the spatial median the object median occurs. This allows one to cut the search space in half. In the cases of small numbers of objects, one can try splitting planes at the limits of each object within the appropriate half and record the maximum. For large numbers of objects, one might try a small set of splitting planes at equally spaced intervals, or even randomly selected intervals, within the appropriate half. Alternatively, a cheap heuristic is to select the splitting plane midway between the object median and the spatial median.

Because of space limitations, we have not dealt with objects spanning the splitting plane. Our results can be extended to handle this case as well, although the analysis is more complicated (MacDonald 1988).

5 Comparisons

Having verified the surface area metric as reasonably accurate, different construction techniques for space subdivision were investigated. Four new construction algorithms, as well as Kaplan's algorithm, were implemented for purposes of comparison and evaluation. All algorithms were implemented on bintrees. The construction algorithms consist of two algorithms in which the spatial median is chosen as the splitting plane, two algorithms in which the splitting plane can be in an arbitrary position, and Kaplan's algorithm as a standard of comparison. These algorithms are the following.

Kaplan's algorithm (zero degrees of freedom in the splitting plane selection). This is simply Kaplan's algorithm with a threshold value of one. Nodes are subdivided until they contain zero or one objects, in a breadth-first order. The maximum height of the tree was set to 30, which was felt to be large enough not to restrict the growth, yet provide a practical bound.

Arbitrary acyclic (two degrees of freedom). Splitting planes can be anywhere within the node, and a node may be divided along any of the three axes. The optimal splitting plane is determined by sampling at nine equally spaced intervals within the node, recording the maximum value of the function given previously. A node is subdivided along whichever axis provides the greatest gain and nodes are subdivided according to highest gain. (Nine is an arbitrary number chosen to approximate the optimal splitting plane, yet not incur unreasonable amounts of computation by finding it exactly during the simulation. We believe that the 10% accuracy achieved by this is sufficient for purposes of this study.)

Arbitrary cyclic (one degree of freedom). Same as arbitrary acyclic, except that the first level of subdivision always occurs along the x axis, the second along the y axis, the third along the z axis, cycling through the three axes.

Spatial median acyclic (one degree of freedom). Same as arbitrary acyclic, except that the spatial median is always chosen as the splitting plane.

Spatial median cyclic (zero degrees of freedom). Same as arbitrary cyclic, except that the spatial median is always chosen as the splitting plane.

These algorithms were encoded as simply as possible without any attempt to optimize the code. It was felt that it was more important that the code be correct, and our emphasis was verification, rather than efficiency. Statistics on the trees were recorded during the construction of the tree. The statistics include the number of interior nodes, the number of empty leaves, the number of nonempty leaves (containing one or more objects), the estimated number of leaves visited, estimated number of interior nodes visited, and the estimated number of objects tested for intersection.

The ultimate goal of the strategies for building the space subdivision structures is to improve performance in actual ray-tracing systems. The performance should therefore be evaluated with scenes that represent a reasonable sample of all scenes subjected to ray tracing. Five scene types proposed by Kingdon (1986) were used. The object distributions are based on three simple random number generators: U^3, which selects a random point within a unit sphere; U^0, which selects a random point on the unit sphere; and U^e, which returns the output of U^0 scaled by a gaussian distributed random number with a mean of 0 and variance of 1. The five scene types used in the simulations were the following.

Small spherical. A set of triangles whose first vertices are U^3 distributed in space and whose other two vertices are $0.010 \cdot U^0$ distributed offsets from the first point.

Large spherical. A set of triangles whose first vertices are U^3 distributed in space and whose other two vertices are $0.333 \cdot U^0$ distributed offsets from the first point.

Small gaussian. A set of triangles whose first vertices are $0.333 \cdot U^e$ distributed in space and whose other two vertices are $0.010 \cdot U^0$ distributed offsets from the first point.

Large gaussian. A set of triangles whose first vertices are $0.333 \cdot U^e$ distributed in space and whose other two vertices are $0.333 \cdot U^0$ distributed offsets from the first point.

Three random vertices. A set of triangles whose vertices are U^3 distributed in space, creating a set of dense, interpenetrating triangles.

The small spherical and small gaussian scenes contain triangles that are roughly $\frac{1}{200}$ times the width of the scene, while the large spherical and large gaussian scenes contain triangles approximately one-sixth the width of the scene, attempting to simulate the limits of object sizes in typical scenes. The gaussian distributions provide a cluster of objects, while the spherical distributions provide more spread out objects. Six instances of each scene were used, varying only in the number of objects comprising the scene. The numbers used were 256, 512, 1024, 2048, 4096, and 8192. The maximum number of nodes was set according to the amount of time and memory required and ranged from 2000 to 8000 nodes, depending on the scene type. Also, for some scene types, only the first five scene sizes were used, to limit computer usage.

Data from the simulations were analyzed to compare the various algorithms. Figure 2 shows a graph of the results for 1024 small spherical objects. The other cases were similar, but are omitted from this paper due to space limitations

In summary, the estimated number of nodes and leaves visited for a given scene were very similar over all five algorithms, as is evident from examining their graphs. Overall, the arbitrary acyclic algorithm performed slightly better than the rest in terms of number of nodes and leaves visited. However, the number of objects intersected varied widely over the different construction algorithms. For this reason and because the object cost is typically higher than the other two costs, let us concentrate on the number of objects intersected in order to evaluate the algorithms' performance.

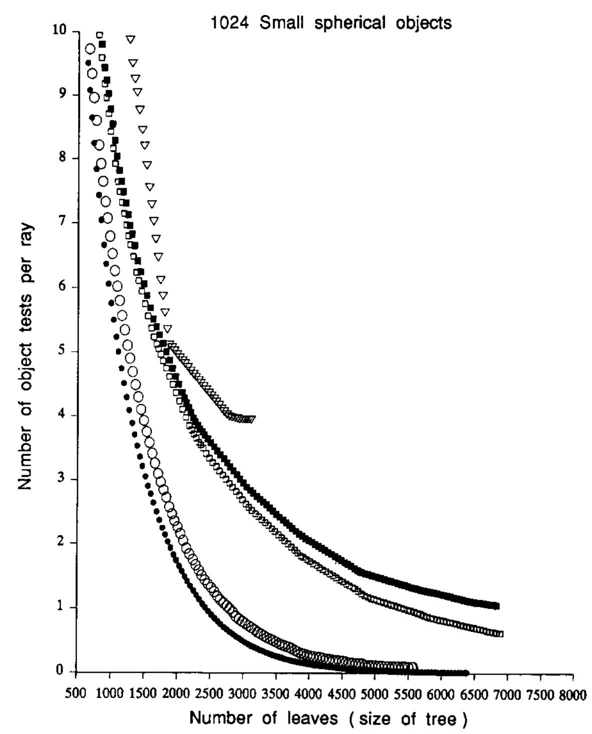

Fig. 2. Summary of simulations. ●, Arbitrary acyclic; ○, arbitrary cyclic; □, spatial median acyclic; ■, spatial median cyclic; ▽, Kaplan (spatial median)

For the small spherical and small gaussian scene types, the arbitrary acyclic algorithm performed the best, providing up to three orders of magnitude reduction in the number of objects tested for intersection. For the large spherical and large gaussian scene types, the arbitrary acyclic algorithm was also the best, but only up to one order of magnitude better. However, for the scenes consisting of three random vertices, the Kaplan method performed best. The general rule seems to be that the arbitrary acyclic algorithm performs best for scenes with nonoverlapping small objects, while Kaplan's performs best for denser scenes with interconnected objects.

The explanation for this behavior is that the arbitrary acyclic algorithm is a greedy algorithm, governing the subdivision by only looking one step in advance. If subdividing a node is not immediately advantageous, then it is not subdivided, even if subjecting the node to two levels of subdivision would be advantageous. Kaplan's algorithm, by virtue of its breadth-first nature and an inability to evaluate the benefit of subdividing a node, may subdivide a node many times, resulting in a gain where the arbitrary acyclic algorithm would not.

These observations indicate that a hybrid of the arbitrary acyclic and Kaplan's algorithms might provide optimum performance in all scene types. A hybrid implementation was performed in which the arbitrary acyclic algorithm was applied to a node first to determine an optimum splitting plane. If it does not find a speed gain above a certain threshold dependent upon the surface area of the node, then the spatial median is chosen. The coordinate is dependent on the level of the node, similar to Kaplan's method except that nodes are only subdivided with one level of subdivision at a time (rather than three levels). This forces the algorithm to assume that subdividing a node results in a decrease in cost, even if the one-step look ahead indicates an increase. Thus, a node that the original algorithm does not find advantageous to subdivide may be subdivided by the hybrid algorithm, resulting in a tree with a higher cost than if the node remained a leaf. The children of this node may then be subdivided, possibly resulting in an overall decrease in the cost of the tree.

This process is used, as in the other algorithms, only to determine the splitting plane, splitting coordinate, and estimated gain if the node were to be subdivided. The selection of the next node to subdivide is, as in the arbitrary acyclic algorithm, the node that has the highest estimated gain. When the hybrid algorithm resorts to selecting the spatial median, the gain associated with this split is set at the threshold, rather than the actual value, which would be lower. This hybrid algorithm was run on each of the five scene types containing 1024 objects, except for the scene type containing three random vertices, which had only 64 objects for efficiency. It performs better overall than any of the other algorithms (it was outperformed slightly by the arbitrary acyclic algorithm in the case of a large gaussian scene).

It is interesting to note that the portions of the graphs pertaining to Kaplan's algorithms often contain line segments and abrupt changes of slope. These are due to the fact that after some point in the construction of the tree, Kaplan's algorithm essentially builds the tree level by level. The line segment portions correspond to individual levels, and the abrupt changes in slope correspond to the filling of a level.

At the end of each simulation, the total number of object instances (number of objects stored at the leaves) was recorded. The arbitrary algorithms produced near optimum numbers, that is, only 10% or 20% more object instances than objects, while Kaplan's and the other two spatial median algorithms produced trees with up to ten times as many object instances as objects. The reason for this is the implicit motivation to keep objects in as few leaves as possible, provided by the cost function used in selecting the splitting plane for arbitrary subdivision.

6 Storage

The simplest and most obvious method of storing the bintree (octree) is as an explicit tree with two (eight) pointers per node. This has a large space requirement, motivating the more compact octree schemes of Glassner and ARTS.

The storage method has a marked effect on the speed of traversing a tree. In ray tracing the internal nodes of a space subdivision are not interesting. All useful information is in the leaves. The traversal cost can be decreased by storing links to neighbors on each of the six faces of each leaf. Samet (1984) describes such links in quadtrees, called *ropes*, and credits Hunter and Steglitz for their invention. A more recent paper also discusses neighbor-finding (Samet and Webber 1988). For the purposes of the following discussion, let each face of each leaf have exactly one neighbor, defined as the smallest node (interior or leaf) whose voxel's surface totally encloses the face of the leaf in question. By this definition, the neighbors of a leaf are not necessarily leaves. However, this definition guarantees that each leaf has exactly one neighbor per face (except leaves on the boundary of the scene, which have none for outer faces).

During traversal of the structure it is necessary to determine the face exited. The neighbor link of a face is followed and if the neighbor is a leaf, processing of the objects within the leaf is performed. If the neighbor is an interior node, then the exit point of the current leaf must be computed and used to descend the neighbor's subtree to find the appropriate leaf. This strategy eliminates all upward traversal of the tree and some downward traversal. In general, when a ray travels from one area to an area of equal or lower subdivision, the neighbor is a leaf and the hierarchy traversal cost is zero. It is only when traveling to an area of higher subdivision that there is any hierarchy cost. In this case the cost is less than the corresponding cost of the methods described earlier because the upward traversal to the common ancestor is eliminated and some of the downward traversal may also be avoided (about equal to the upward traversal eliminated). Therefore, the neighbor links reduce the hierarchy cost significantly, at the added expense of six pointers per leaf.

A further modification of the neighbor links is to redefine the neighbors of a face as all leaves adjacent to that face. Now, all neighbors are leaves, but any given face may have more than one neighbor, which requires more memory per leaf than the previous link strategy. However, in the case of spatial median subdivision, the amount of memory required is now less than 12 pointers per leaf on average, only twice that of the former method. The average of 12 pointers per leaf stems from the observation that, although some faces have a large number of neighbors, others have only one neighbor, with the average being two pointers per face. This is illustrated in Fig. 3, which shows $n+1$ faces, and $2 \cdot n$ links, and hence $\frac{2 \cdot n}{n+1}$ links per leaf, which means fewer than two pointers per face. With arbitrary subdivision, the number of pointers per face may be higher, because Fig. 3 no longer covers all possible subdivision cases.

The storage of the neighbors for a leaf consists of six integers representing the number of neighbors of each face, plus a list of pointers to the neighbors of each face. Alternatively, the neighbors could be stored in a two-dimensional bintree (or quadtree) to quickly determine the appropriate neighbor for a given exit point. In fact, such a quadtree is implicit in an octree already. A single neighbor link to a node that has further subdivision is sufficient to find all adjacent neighbors because the standard quadtree search can be performed on the octree structure simply by ignoring the coordinate whose value is known (i.e., if the neighbor is being sought across a "positive" x face, then only the "negative" x nodes in the octree are relevant).

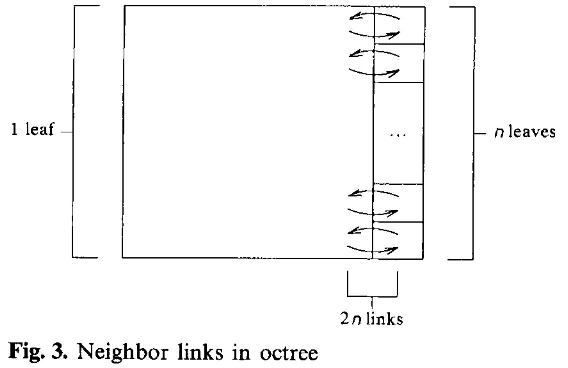

Fig. 3. Neighbor links in octree

The *complete neighbor links* scheme eliminates the hierarchical traversal altogether, because finding the next node only requires following the links, but it introduces the additional cost of determining which link to follow if a leaf has more than one neighbor on a given face. We assume that the number of neighbors of a leaf is proportional to its surface area.

Better search performance may result from the use of a two-dimensional bintree to search for the neighbors or by performing a binary search on the sorted neighbor lists. Either of these two methods reduces the expected number of tests per face to $\log n$ complexity. The form of the tests is single comparisons in the case of the two dimensional bintree, rather than four comparisons. The expected number of comparisons is therefore proportional to

$$\sum_{l=1}^{N_l} SA(l) \log SA(l)$$

$$\sim \sum_{l=1}^{N_l} \frac{1}{N_l^{2/3}} \cdot \frac{-2}{3} \cdot \log N_l \sim \sqrt[3]{N_l} \cdot \log N_l$$

Although it appears that the neighbor links approach may have large space requirements, there is a memory-speed tradeoff that can be invoked. Instead of defining links to occur at all leaves in the tree, one can define the links to occur at all interior nodes that only have leaves for children. This decreases the extra space to approximately one-eighth of the original space requirements in the octree case, or one half in the case of a bintree. This method incurs the same traversal cost as the original neighbor links plus one additional upward traversal per leaf and possibly one downward traversal.

Table 2. Number of parent-to-child and child-to-parent traversals recorded from the simulation

Scene type	Up/down traversals, 1000 nodes		
	Up	Down	Neighbors down
1000 Small spherical	36.35589981	36.38169861	9.951199532
1000 Large spherical	15.85369968	20.09070015	8.987500191
1000 Small gaussian	33.94269943	33.94810104	10.09840012
1000 Large gaussian	24.50469971	25.91119957	8.954000473
64 3-Random verts	15.17660046	19.25169945	9.043399811

More generally, the linking can be defined only for the set of nodes at a particular height above the leaves. For example, links may be stored in all nodes that are a fixed distance n above the deepest leaf in their subtree. The case $n=1$ corresponds to the above method of storing at all nodes that only have leaves for children. The amount of memory required is proportional to $(\frac{1}{8})^n$ in the case of an octree, yet the extra traversal cost is only proportional to n. A suitable value of n results in an appropriate tradeoff between space and the additional up and down traversals. For practical cases n can be chosen so that the extra indirection to follow links is modest and the additional storage for links is vanishingly small.

A neighbor links strategy was implemented, using the simple definition of neighbors which gives exactly one neighbor per face, as opposed to the complete neighbor links strategy. One instance of each of the five scene types was used to build an arbitrary acyclic type bintree, with the neighbor links for each leaf computed. All scenes had 1000 objects and the bintrees constructed contained 1000 nodes. After building the bintrees, 10000 random rays were traced and the number of parent-to-child and child-to-parent movements were recorded for each of the conventional traversal algorithms and the neighbor links method. These numbers indicate the savings in traversal cost by using the neighbor links strategy.

Table 2 summarizes the number of parent-to-child and child-to-parent traversals recorded from the simulation. The second and third columns give the number of up and down links followed for the conventional traversal algorithms. The fourth column gives the number of down links followed for the neighbor link algorithms (there are no up links followed). If it is assumed that the cost of a single upward traversal is equivalent to a single downward traversal, then these numbers show that the neighbor link scheme decreases the traversal cost to between one-seventh and one-quarter of the cost of an ARTS-type traversal method.

Storage of the lists of objects that belong in each leaf have large space requirements. Glassner stores all the object lists in a single array of object indices, where each list ends with a "nil" index. Glassner's scheme provides a separate object list for each leaf. A more compact scheme would allow more than one leaf to point to the same object list. In cases where there are many duplicate leaf lists, this scheme would result in significant memory savings. There would be an added cost during the traversal phase in order to identify duplicate lists but only one extra level of indirection. Even more savings would result if lists that are subsets of other lists are identified, and a pointer to the beginning of a sublist within a larger list used to avoid explicit storage of the sublist. The larger list would have to be organized so that the sublist is at the end.

The most compact scheme is to partition the set of objects into equivalence classes, where each equivalence class is a set of objects that belong in the same set of leaves. In the worst case, each equivalence class consists of one object, in which case this scheme is equivalent to the above many-to-one linking with the overhead being a single extra level of indirection. The object list for a leaf is thus a list of equivalence classes, rather than a list of object indices. Although the computation of the equivalence classes might be quite expensive, it is only computed once when the space subdivision is constructed. The savings in space might well outweigh the extra computing time.

7 Discussion

The cost of ray tracing using space subdivision trees can be estimated by the number of interior nodes, leaves, and objects visited per ray, and the respective costs of these visits. This paper reports new construction algorithms which represent considerable improvement over conventional methods in terms of reducing the number of nodes, leaves, and objects visited by a ray. The algorithms employ the surface area heuristic and a heuristic for estimating the optimal splitting plane located between the spatial median and the object median.

The efficiency of traversal has been improved by attacking its two main costs, the processing of interior nodes (a major improvement) and the computation of the ray exit point (a minor improvement). The neighbor link strategy has been employed to significantly reduce the number of interior nodes visited compared to Glassner's algorithms.

Many of the ideas in this paper should carry over to hierarchical extent trees. All of the ideas should be examined with respect to higher-dimensional data structures, dynamic data structures, and multiprocessor algorithms. We suggest a few areas for future research in our closing remarks.

In computer animation, it is common for scenes to change from frame to frame, as objects appear, disappear, and change position, shape, color, and other attributes. The data structures representing the scene must be updated to reflect these changes. An important issue when choosing a data structure to represent scenes is whether the structure allows *dynamic* modification as the scene changes, and whether the dynamic modification is more efficient than rebuilding a *static* structure each time the scene changes. The restriction to static structures is not unreasonable, as static structures are appropriate in cases where the viewpoint changes often compared to the objects in the scene. But when this is not the case, our algorihms must be extended to accommodate dynamic changes. One specific method of dealing with dynamic objects is to treat time as simply another dimension, with the data structure subdividing the objects in 4-space. Glassner (1988) has reported on such an approach.

Our discussion has not addressed issues related to multiprocessors. Other authors have suggested a variety of techniques for utilizing multiprocessors in ray tracing. We believe that many of our techniques can be applied here as well.

Acknowledgements. This work was supported by an operating grant and a postgraduate scholarship from the Natural Sciences and Engineering Research Council of Canada and by equipment and operating funds from Digital Equipment of Canada.

References

Amanatides J, Woo A (1987) A fast voxel traversal algorithm for ray tracing. Proc Eurographics '87:1–10

Arnaldi B, Priol T, Bouatouch K (1987) A new space subdivision method for ray tracing CSG modeled scenes. Visual Computer 3:98–108

Cleary JG, Wyvill G (1988) Analysis of an algorithm for fast ray tracing using uniform space subdivision. Visual Comput 4:65–83

Cook RL, Porter T, Carpenter L (1984) Distributed ray tracing. Comput Graph 18: 137–145

Fujimoto A, Tanaka T, Iwata K (1986) ARTS: accelerated ray-tracing system. IEEE Comput Graph Appl 6:16–26

Glassner AS (1984) Space subdivision for fast ray tracing. IEEE Comput Graph Appl 4:15–22

Glassner AS (1987a) An overview of ray tracing. SIGGRAPH '87 Introduction to Ray Tracing Course Notes

Glassner AS (1987b) Spacetime ray tracing for animation. SIGGRAPH '87 Introduction to Ray Tracing Course Notes

Glassner AS (1988) Spacetime ray tracing for animation. IEEE Comput Graph Appl 8:60–70

Goldsmith J, Salmon J (1987) Automatic creation of object hierarchies for ray tracing. IEEE Comput Graph Appl 7:14–20

Heckbert PS (1982) Color image quantization for frame buffer display. Comput Graph 16:297–307

Kaplan MR (1985) The uses of spatial coherence in ray tracing. SIGGRAPH '85 Course Notes no 11

Kay TL, Kajiya JT (1986) Ray tracing complex scenes. Comput Graph 20:269–277

Kingdon SJ (1986) Speeding up ray-scene intersections. Thesis, Univ Waterloo

MacDonald JD (1988) Space subdivision algorithms for ray tracing. Thesis, Univ Waterloo

Rubin SM, Whitted T (1980) A three-dimensional representation for fast rendering of complex scenes. Comput Graph 14:110–116

Samet H (1984) The quadtree and related hierarchical data structures. Comput Surv 16:187–260

Samet H, Webber RE (1988) Hierarchical data structures and algorithms for computer graphics. IEEE Comput Graph Appl 8:48–68; 8:59–75

Scherson ID, Caspary E (1987) Data structures and the time complexity of ray tracing. Visual Comput 3:201–213

Stone L (1975) Theory of optimal search. Academic Press, New York

Weghorst H, Hooper G, Greenberg DP (1984) Improved computational methods for ray tracing. ACM Trans Graphics 3:52–69

DAVID MACDONALD received his BSc degree in mathematics and computing sciences from St. Francis Xavier University in 1986 and his MMath degree in computer science from the University of Waterloo in 1988. He is currently a software engineer at Visual Edge Software Ltd., Montreal, Quebec. His research interests include data structures, ray tracing, and interactive graphics for scientific visualization with an emphasis on the analysis of continuous fields.

KELLOGG S. BOOTH is professor of computer science and director of the Institute for Computer Research at the University of Waterloo, where he has been on the faculty since 1977. Prior to that he was a member of the research staff in the Computation Department of the Lawrence Livermore National Laboratory in the computer graphics group. His research interests include high performance graphics workstations, computer animation, user interface design and analysis of algorithms. He received his BS in mathematics from Caltech in 1968 and his MA and PhD in computer science from UC Berkeley in 1970 and 1975. He is a member of the Canadian Man-Computer Communications Society, IEEE and ACM, and is a past chairman of ACM SIGGRAPH. Dr. Booth is a consultant to government and industry on computer graphics and related areas of computer science.