6 A Survey of Ray Tracing Acceleration Techniques

JAMES ARVO AND DAVID KIRK

1 INTRODUCTION

One of the greatest challenges of ray tracing is efficient execution. Despite its impressive repertoire, ray tracing is often dismissed as being too computationally exorbitant to be useful. Efficiency is therefore a critical issue and has been the focus of much research from the beginning. This has led to creative approaches involving novel data structures [2, 33, 37, 52, 56,], numerical methods [32, 58, 59, 62], computational geometry [13, 44], optics [54], statistical methods [10, 15, 42, 48], and distributed computing [7, 14, 22, 41, 45, 60] among many others. A would-be implementer now has a tremendous assortment of techniques to choose from and many considerations to balance, some of which are listed in *Figure 1*. Nearly all these techniques give rise to useful combinations, further increasing the possibilities.

Though this area is still undergoing rapid development, it is a worthwhile exercise to examine what has been done. As Sutherland *et al.* [57] demonstrated in their characterization of ten hidden-surface algorithms, identifying a taxonomy of current methods can sometimes provide a perspective from which new approaches become apparent. Since several important themes have emerged in the area of efficient ray tracing, the time is ripe for such a taxonomy. Toward this end we attempt to unify some of the terminology and methods which have evolved independently yet build upon similar concepts.

One shortcoming of this survey is the absence of quantitative comparisons. The information contained herein is insufficient to make a clear and absolute exhaustive ray tracing is by far the most intuitive solution, and it continues to play a role in the processing of subproblems within more complicated techniques.

3 A BROAD CLASSIFICATION

Faced with the task of accelerating the process of ray tracing, there are three very distinct strategies to consider: (1) reducing the *average cost* of intersecting a ray with the environment, (2) reducing the *total number* of rays intersected with the environment, and (3) replacing individual rays with a more general entity. These appear in *Figure 2* as 'faster intersections,' 'fewer rays,' and 'generalized rays,' respectively. The category of 'faster intersections' further separates into the subcategories of 'faster' and 'fewer' ray–object intersections. The former consists of efficient algorithms for intersecting rays with specific primitive objects, while the latter addresses the larger problem of intersecting a ray with an environment using a minimum of ray–object intersection tests. The distinction between these two subcategories is blurred

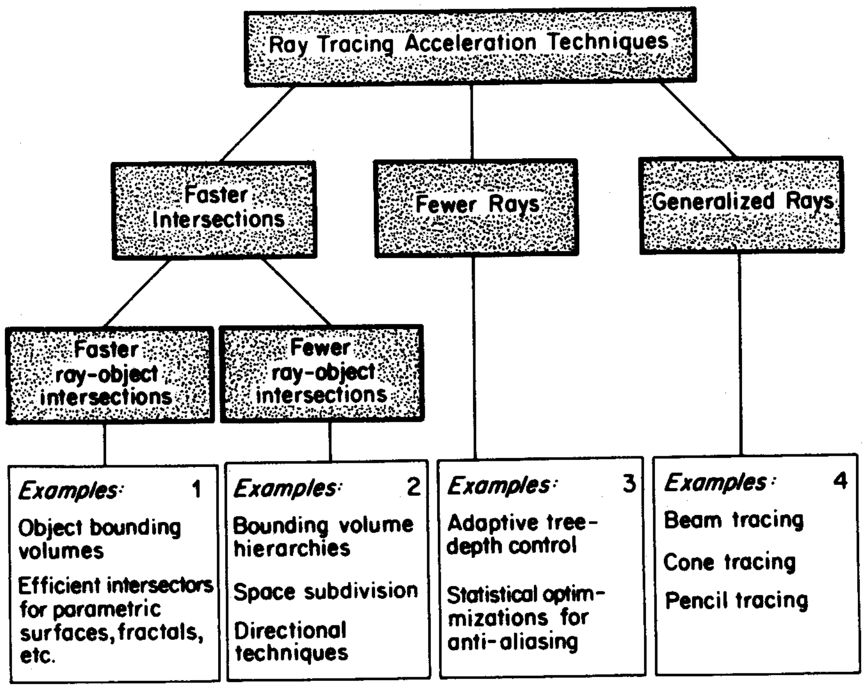
Ray Tracing Acceleration Techniques
Faster Intersections
Fewer Rays
Generalized Rays
Faster ray-object intersections
Fewer ray-object intersections
Examples: 1
Object bounding volumes
Efficient intersectors for parametric surfaces, fractals, etc.
Examples: 2
Bounding volume hierarchies
Space subdivision
Directional techniques
Examples: 3
Adaptive tree-depth control
Statistical optimmizations for anti-aliasing
Examples: 4
Beam tracing
Cone tracing
Pencil tracing

Fig. 2. A broad classification of acceleration techniques.

somewhat by algorithms which decompose what is normally thought to be a single primitive object into many simpler pieces for the sake of efficiency. An example of this is the approach developed by Sweeney *et al.* for intersecting B-spline surfaces [58]. By subdividing a single surface into easily handled fragments and constructing a bounding volume hierarchy, the algorithm resembles the techniques described in the next section for dealing with collections of autonomous objects.

Other primitive object intersection algorithms are extremely special-purpose, often embodying analytic solutions for the point of intersection with a ray. Though it frequently requires considerable ingenuity to formulate such closed-form expressions, as with algebraic surfaces [28] and Steiner patches [53], these algorithms will not be explored in this survey. Nearly all procedural object approaches have as their basis some technique for efficiently computing the intersection, making the distinction between object definition and acceleration vague as well.

The category labeled 'fewer rays' consists of techniques which allow us to reduce the number of rays which need to be intersected with the environment. This includes first-generation rays as well as those created by reflection, refraction, and shadowing. The first such technique was *adaptive tree-depth control* introduced by Hall *et al.* [27]. Instead of terminating the ray tree at a predefined depth or at nonreflective opaque surfaces, Hall's termination criterion took into consideration the maximum contribution to the pixel color which could result by continuing the recursion. Setting a threshold on this contribution made it possible to eliminate the processing of many rays deep in the ray tree without altering the result perceptibly. This led to considerable savings even for environments with many highly reflective surfaces.

Other techniques for reducing the number of rays are applicable when anti-aliasing through supersampling. By detecting situations in which a relatively small number of samples produce statistically reliable results over some region of the image, many first-generation rays (hence entire ray trees) can be eliminated. Though often thought of as part of the anti-aliasing algorithms, these statistical techniques are, first and foremost, performance optimizations.

The last category, labeled 'generalized rays,' consists of a number of techniques which begin by replacing the familiar concept of a ray with a more general entity which subsumes rays as a special (degenerate) case. For instance, cones of both circular [1] and polygonal [30] cross section have been used successfully. Though the essential concepts of ray tracing remain largely intact, at the heart of these techniques lies the idea of tracing many rays simultaneously. As we shall see in Section 9, this presents many interesting advantages, but limitations as well.

Applicability:

- Does it apply to all rays or just a special class of rays?
- Is it applicable in the context of constructive solid geometry?
- Does it impose a restriction on the class of primitive objects?
- Is it applicable when a temporal dimension is added?

Performance:

- Will it be fast enough to meet the application requirements?
- How well does it scale to very complex environments?
- Does the cost of pre-processing eventually outweigh its benefit?
- How well does it exploit available coherence?

Resources:

- What are the storage limitations of the host machine?
- Can the algorithm make appropriate space/time trade-offs?
- What is the cost of floating point arithmetic relative to integer arithmetic?
- Does the host machine have multiple processors?

Simplicity:

- How difficult is the algorithm to implement?
- How dependent is it on machine architecture?
- Can it extend existing code or does it require a complete rewrite?
- Does it require a priori selection of unintuitive parameters?

Fig. 1. Some of the considerations which affect the choice of acceleration technique(s).

decision about which algorithm is best for a given application. This deficiency is a reflection of the current state of the art, and is due in part to the difficulty of *a priori* performance analysis. Though there is a movement toward quantitative comparisons through standard benchmarks [26], this is not yet widely practiced. Consequently, we shall concentrate on the underlying concepts and build a framework which highlights differences and similarities. We also discuss pitfalls uncovered by experience and identify several unexplored possibilities.

We begin with background material in Section 2, and proceed in Section 3 to classify acceleration techniques into four broad categories. The first of these categories deals with efficient operations on individual geometrical objects. Sections 4, 5, and 6 cover three families of techniques which fall into the second and largest category of acceleration techniques, those which reduce the cost of 'tracing a ray' in the context of complex environments. In Section 7 we discuss coherence, a concept which appears in many guises and is utilized to some degree by all acceleration techniques. The statistical methods in Section 8 fall into the third category of techniques, those which reduce the total number of rays which need to be processed. Section 9 covers the techniques of the fourth category, those which generalize the concept of a ray in order to more efficiently exploit coherence. Sections 10 and 11 describe special optimizations for CSG (constructive solid geometry) and parallel architectures respectively. Finally, in Section 12 we discuss ways in which many of these techniques can be used in unison. With few exceptions, the techniques of each section are discussed in the order of their chronological development.

2 BACKGROUND

The generality of ray tracing is due to its almost exclusive dependence upon a single operation; calculating the point of intersection between a ray in three-space and an atomic geometrical entity, or *primitive object*. Examples of primitive objects include elementary shapes such as polygons, spheres, and cylinders, as well as more complex shapes such as parametric surfaces [58, 59] and swept surfaces [62]. The task we are primarily concerned with in this survey is that of intersecting rays with a large collection of primitive objects defining an *environment*. For each ray this ultimately reduces to computing th point of intersection closest to the ray origin which results from any of ie individual primitive objects in the environment. The cost of this operation typically overshadows everything else, accounting for the vast bulk of the time consumed by ray tracing. An often-quoted statistic reported by Whitted [65] is that better than 95% of the time can be spent performing this operation for complex environments. Despite dramatic algorithmic improvements, the demand for ever increasing complexity tends to keep this figure realistic or even conservative. Therefore, the search for more efficient techniques continues to be a lively topic of research.

Following common practice, we shall limit our discussion to this 'intersection problem' and assume that a negligible amount of time is spent in all remaining tasks, such as shading calculations and common bookkeeping operations. We note at the outset that the intersection problem has a trivial but usually impractical solution which is commonly referred to as 'standard' (or 'traditional') ray tracing. This solution entails intersecting each ray with the environment by simply testing each and every primitive object and retaining the nearest point of intersection (if one exists). This has a time complexity which is linear in the number of objects. We shall refer to this as *exhaustive* ray tracing in preference to the word 'standard,'which tends to imply widespread application or at least a long history as the method of choice. It is far more appropriate to reserve the term 'standard ray tracing' for the *illumination model* introduced by Whitted [65], which is independent of the mechanism for computing ray–environment intersections. Nevertheless,

4 BOUNDING VOLUMES AND HIERARCHIES

The most fundamental and ubiquitous tool for ray tracing acceleration is the *bounding volume* (also known as an *extent* or *enclosure*). This is a volume which contains a given object and permits a simpler ray intersection check than the object. Only if a ray intersects the bounding volume does the object itself need to be checked for intersection. Though this actually increases the computation for rays which come near enough to an object to pierce its bounding volume, in a typical environment most rays closely approach only a small fraction of the objects. The result is a significant net gain in efficiency. Whitted [65] initially used spheres as bounding volumes, observing that they are the simplest shapes to test for intersection.

Used in this way, bounding volumes substitute simple intersection checks for more costly ones, but do not decrease their number. From a theoretical standpoint this may reduce the computation by a constant factor, but cannot improve upon the linear time complexity of exhaustive ray tracing. To alleviate this problem, Rubin and Whitted [51] introduced the notion of hierarchical bounding volumes to ray tracing in order to attain a theoretical time complexity which is logarithmic in the number of objects instead of linear. By enclosing a number of bounding volumes within a larger bounding volume it was possible to eliminate many objects from further consideration with a single intersection check. If a ray did not intersect the *parent volume*, there was no need to test it against the bounding volumes or objects contained within. A hierarchy was formed by repeated application of this principle.

This type of 'logarithmic search' was previously employed by Clark [6] to accelerate clipping during display of hierarchically organized data. If a bounding volume was entirely outside the viewing frustrum, its contents could be immediately rejected, whether it enclosed displayable elements or additional bounding volumes. If a bounding volume was entirely inside the viewing frustrum, all of its descendants could be rendered with no further clipping operations. The relationship between this algorithm and that of Rubin and Whitted is quite close. If we consider a ray to be a degenerate viewing frustrum possessing no interior, the algorithms are virtually identical from the standpoint of hierarchy traversal.

The volumes employed by Rubin and Whitted were rectangular parallelepipeds, more commonly known as *bounding boxes*, which are oriented so as to closely fit their contents and minimize their size. In order to perform the ray–box intersection tests, each ray was first transformed into the coordinate space of the bounding box. This made the subsequent test between the transformed ray and the axis-aligned box very straightforward. The simplicity of this operation motivated the use of bounding boxes for representing the geometry at the terminal nodes of the hierarchy as well. For instance, Rubin

```
procedure BVH__Intersect(in ray, node)
begin
    if node is a leaf then
        Intersect(ray, node.object)
    else if Intersect__P(ray, node.bounding__volume) then
        for each child of node do
            BVH__Intersect(ray, child);
    end
```

Fig. 3. A procedure for intersecting a ray with a collection of objects organized in a bounding volume hierarchy. Procedure 'Intersect' and function 'Intersect__P' hide many of the common low-level details.

and Whitted chose to represent polygons by one or more bounding boxes which were degenerate along one axis.

Figure 3 is an outline of procedure 'BVH__Intersect' which intersects a ray with a collection of objects organized in a bounding volume hierarchy. The data structure for this hierarchy is assumed to be a tree (or more generally a directed acyclic graph, or DAG) with an arbitrary branching factor at each internal node. Thus, bounding volumes may enclose any number of other bounding volumes. Each leaf node of the tree is a single primitive object while each interior node consists of a bounding volume and a list of pointers to other nodes in the tree. The procedure 'Intersect' called from within 'BVH__Intersect' is responsible for invoking the appropriate ray–object intersection procedure for the type of primitive object passed to it, and the 'ray' parameter encodes a 3-D origin, a direction vector, and a *distance interval*. Points of intersection which fall outside the distance interval (measured along the ray) are to be ignored. We will assume that 'Intersect' observes this rule because it simplifies this and subsequent examples. In addition, when a new point of intersection is found, 'Intersect' is assumed to shrink the far end of the distance interval to that point and save away whatever additional information will be needed for shading, such as the surface normal. These conventions hide some of the common mechanisms, such as identifying the closest intersection among several candidates, and therefore allow us to concentrate on the more important algorithmic features.

The function 'Intersect__P' (where the 'P' stands for predicate) is very similar to 'Intersect' except that it returns a boolean value indicating whether an intersection was found and it does *not* alter the ray's distance interval. This function is used exclusively to determine if a bounding volume is hit by a ray, whereas the automatic adjustment of the distance interval is only appropriate for true object intersections.

Given 'Intersect' and 'Intersect__P,' the task of intersecting a ray with a given bounding volume hierarchy is quite straightforward. The process begins with the root node of the tree, representing a bounding volume enclosing the entire environment, and an 'unbounded' ray, that is, one whose distance interval is zero to 'infinity.' Each recursive reference of 'BVH_Intersect' descends another level of the hierarchy, and the recursion terminates with ray–object intersection tests at the leaves. At each level, the ray is tested against all the sibling bounding volumes and we only descend into the ones which are hit by the ray. The others are not processed any further, allowing us to *prune* the branches which they enclose.

An additional benefit of adjusting the ray's distance interval in the way we have described is that it performs a useful optimization [19, 50]. Once a point of intersection has been found with some object, and an upper bound placed on the distance interval, all objects or bounding volumes which intersect the ray completely beyond this bound can be ignored. This provides a second mechanism by which branches can be pruned from the hierarchy during the processing of a ray. An example of this is shown in *Figure 4*. If bounding volume V_1 is processed before V_2, the contents of the latter need not be tested because the point of intersection with object O_1 is closer than any which might occur within V_2. This saves at least one ray–object intersection test and potentially many in cases where V_2 encloses other bounding volumes. If sibling bounding volumes are processed in some fixed order (e.g via a static linked list or array), this technique will take advantage of fortuitous instances in which a nearby intersection is found early on. In Section 4.4 we describe an algorithm introduced by Kay and Kajiya [37] which uses a sorting operation to more consistently benefit from this optimization.

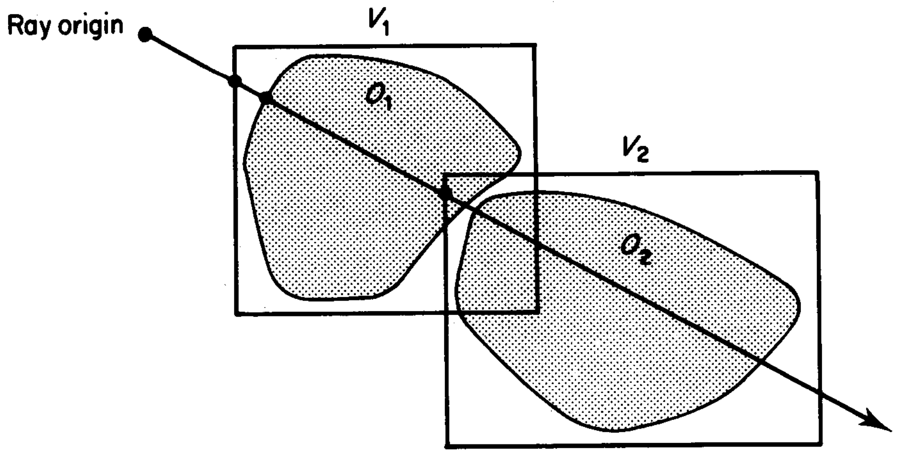

Fig. 4. An optimization which results from shrinking the distance interval associated with a ray whenever an intersection is found. The contents of volume V_2 need not be tested against this ray if the intersection with object O_1 is found first.

4.1 Heuristics for Bounding Volume Optimization

To further improve the efficiency of bounding volumes, Weghorst *et al.* [63] investigated the trade-offs between two competing factors: tightness of fit and cost of intersection. By selecting a sphere, box, or cylinder depending on the characteristics of each object (or cluster of objects) to be enclosed, they were able to increase the efficiency of individual and hierarchically organized bounding volumes. The criterion for this selection began with the observation that the total computational cost associated with an object and its bounding volume is given by

$$\text{Cost} = n * B + m * I \tag{1}$$

where n is the number of rays tested against the bounding volume, B is the cost of each test, m is the number of rays which actually hit the bounding volume, and I is the cost of intersecting the object within. Assuming both n and I are fixed, we would like to select a bounding volume which is both inexpensive, making B small, and as tight fitting as possible, minimizing m. One must usually settle for a compromise, however, and making the right trade-off requires estimating both cost and fit. Weghorst *et al.* used the enclosed volume as a measure of fit, observing that it is related to the *projected void area* with respect to any direction; that is, to the difference in the projected areas of the bounding volume and the enclosed object. This difference in area indicates how likely a ray is to hit the bounding volume without hitting the enclosed object. A large void area, resulting from a loose fit, can increase m and cause many unnecessary object intersection checks. Reducing m even at the expense of an increase in B is sometimes warranted. Weghorst *et al.* introduced a simple heuristic to determine when such a trade-off is likely to be advantageous. First, each type of bounding volume was assigned a relative complexity factor to rank the computational cost of the ray intersection tests. In their implementation, spheres were given the lowest complexity rating and cylinders the highest. Then, each volume was 'tried' in turn as a potential bound, and the one producing the smallest *product* of volume and complexity factor was selected. This applies equally well to the bounding volumes of the internal nodes of a hierarchy. Because this heuristic did not take the complexity of the enclosed object into account, however, an interactive program was used to occasionally override the algorithmically selected bounding volume.

Figure 5 shows a number of possible bounding volumes for a complex object, perhaps a surface of revolution. The shaded region represents the projected void area. In most instances this void area is dependent upon the direction along which we form the two-dimensional projection. Assuming for

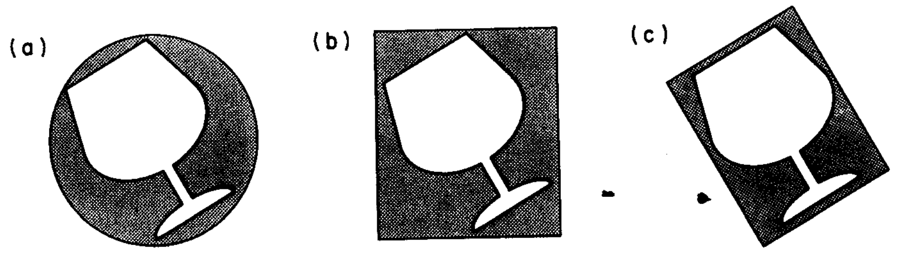

Fig. 5. A comparison of three different types of bounding volumes for the same primitive object. Each presents a different cost/fit ratio. (a) Bounding sphere. (b) Axis-aligned bounding box. (c) Oriented bounding box.

simplicity that the rays which we are tracing through the environment are effectively randomized by multiple reflections and refractions, the average projected void area (over all directions) becomes the relevant measure of fit. As we shall see in the following section, the surface area of the bounding volume is closely related to this average.

Volumes (b) and (c) in *Figure 5* are axis-aligned and transformed (oriented) bounding boxes, respectively. The latter clearly produces a better fit in this case but carries with it the extra cost of a ray transformation for every ray-bounding volume intersection check. Hence, these are effectively different types of bounding volumes because they present different cost/fit trade-offs. In the case of a complex object, however, the relatively small additional cost of the ray transformation in the bounding volume intersection test may be paid back many times over through a significant reduction in number of ray–object intersection tests. This type of transformation can also be applied to other types of bounding volume; to orient cylinders for example, or to deform spheres into ellipsoids.

Another strategy for achieving a better fit is to use multiple bounding volumes for a single object. For instance, we can enclose the object within the

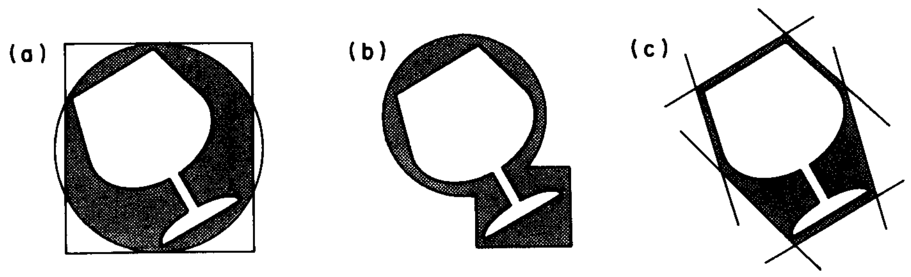

Fig. 6. The intersection and union of multiple bounding volumes can be used to obtain a better fit. Each approach requires a different ray-intersection algorithm for best performance. (a) Intersection of box and sphere. (b) Union of box and sphere. (c) Intersection of slabs.

intersection of two or more bounding volumes, as in *Figure 6(a)*. A ray must then intersect *all* the volumes before the enclosed object needs to be tested. The cost of this composite bounding volume is the sum of the individual volume costs in the case of a ray which hits them all, but is only the cost of the 'first' volume in the case of a distant miss. Alternatively, the object may be covered by the union of two or more bounding volumes, as in *Figure 6(b)*. Here the object must be tested if *any* of the bounding volumes are hit, making the cost of a complete miss more expensive in this case. Finally, *Figure 6(c)* shows a bounding volume created by the intersection of infinite slabs. This type of bounding volume will be discussed in Section 4.3.

4.2 Predicting the Effectiveness of a Hierarchy

In order to better predict the effectiveness of a bounding volume we need to have information about the distribution of rays which will be tested against it. If every ray were to hit the enclosed object, no bounding volume would be beneficial. That is, every type of bounding volume, no matter how simple, would only increase the average cost of the intersection tests. On the other hand, if no ray even approaches the enclosed object, any bounding volume which is less expensive to test than the object is an advantage. In such a case the cheapest bounding volume is the best, independent of any other factor. In most situations the collection of rays tested against a given bounding volume falls somewhere between these two extremes, and in this mid-ground fit becomes a relevant factor as well as cost.

One way to extract useful information about ray distributions is to consider the effect of one bounding volume upon another instead of examining them in isolation. In particular, we will examine how one bounding volume affects the distribution of rays seen by one or more bounding volumes nested completely within it.This leads to a very natural way of predicting the performance of an entire bounding volume hierarchy. Following the approach of Goldsmith and Salmon [23] we consider the conditional probability of a ray hitting an inner volume, B, given that it has hit a surrounding volume, A. See *Figure 7(a)*. The standard notation for this conditional probability is $Pr(r \text{ hits } B \mid r \text{ hits } A)$, where r is a 'random' ray. For simplicity we assume that all rays which hit A are 'equally likely' (i.e. uniformly distributed). Even though the distribution of rays is usually far from uniform in practice, this scenario nevertheless gives a more realistic picture with respect to B than if we had considered it in isolation. The conditional probability expresses the important fact that A 'filters out' most of the rays which would not have hit B.

Under this randomness assumption, a simple calculation shows that $Pr(r \text{ hits } B \mid r \text{ hits } A)$ is equal to the ratio of the average projected area of B to the average projected area of A. This is quite convenient because the average

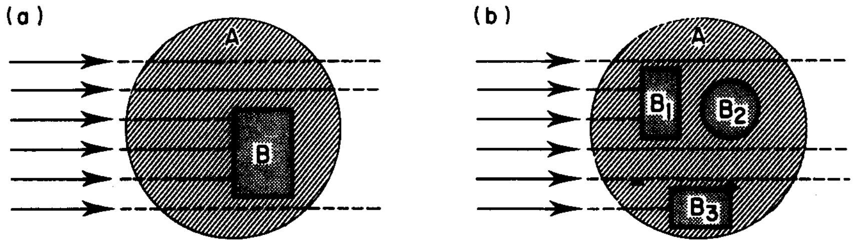

Fig. 7. (a) We wish to compute the conditional probability of a ray hitting B given that it has hit A. This can be used in cases like (b) to compute the average cost of intersecting a ray with the arbitrary contents of a bounding volume.

projected area of a convex body is equal to one quarter of its surface area [61, p. 110]. Therefore, if both A and B are convex (as most bounding volumes are), we have that

$$Pr(r \text{ hits } B \mid r \text{ hits } A) = \frac{< P(B, \mathbf{d}) >}{< P(A, \mathbf{d}) >} = \frac{S(B)}{S(A)} \tag{2}$$

where $P(V, \mathbf{d})$ is the projected area of V along direction $\mathbf{d}$, $< >$ means the average taken over all directions $\mathbf{d}$, and $S(V)$ is the surface area of volume V. This relationship will be the key to analyzing the expected cost of hierarchically arranged bounding volumes.

There are two relevant costs associated with a bounding volume within a hierarchy: (1) the fixed cost of a ray intersection test with the volume itself, and (2) the average cost of a ray intersection test with its contents given that that ray has hit the volume (*Figure 7(b)*). We shall call these the *external cost* and the *internal cost*, denoted EC and IC, respectively. These correspond to the constant B and a generalization of the constant I in equation (1). Given equation (2) for conditional probability we can compute the (average) internal cost of a bounding volume, A. There are two components to this cost, and they can be expressed in terms of the costs of the enclosed items. First, there is the fixed cost of testing a ray against all of A's immediate children. This must be done for every ray which hits A. There is also the cost of testing the contents of the children which are actually hit by a given ray (neglecting the distance interval optimization). The former are external costs of the children and the latter are internal costs of the children weighted by the conditional probability that they are hit. This is expressed by equation (3) for a bounding volume, A, enclosing child volumes $B_1, B_2, \ldots B_n$.

$$IC(A) = \sum_{i=1}^{n} \left\{ EC(B_i) + \frac{S(B_i)}{S(A)} * IC(B_i) \right\}. \tag{3}$$

By definition the internal cost of a primitive object is zero, and this implies that we needn't know its surface area. Recursive application of equation (3) results in an average cost of intersecting a ray with a given bounding volume hierarchy expressed in terms of surface areas and external costs. This is valid regardless of the types of primitive objects or bounding volumes used, provided we have a measure of their surface areas and costs of intersection (i.e. their external costs).

We hasten to point out that equation (3), though based on important geometric relationships among bounding volumes, is still a heuristic and not an infallible measure of cost. Many simplifying assumptions have been made in order to arrive at this convenient equation. In addition to the proper nesting, convexity, and randomness assumptions noted earlier, an implicit assumption has been that the external cost of a bounding volume is constant for all rays and independent of whether or not the volume is hit by the ray. We have also neglected the effects of objects occluding one another. For example, in *Figure 7(b)* any of the rays shown which hit an object within B_1 need not be tested against the contents of B_2 due to the distance interval optimization. Occlusion such as this serves to increase efficiency slightly above that which is predicted by equation (3). However, nonuniformity in the distribution of rays can be far more significant and can drastically change the actual cost in either direction from the predicted value.

4.3 Constructing a Hierarchy

Constructing a bounding volume hierarchy involves two types of decisions: which clusters of objects (or bounding volumes) to enclose and what type of bounding volume to enclose them with. In Section 4.1 we described one heuristic for selecting the volume type, and in Section 4.2 we derived an expression for predicting the effectiveness of a given hierarchy. We now turn to the problem of selecting the clusters of objects or bounding volumes when constructing the hierarchy initially. This is a challenging problem because the number of possible hierarchical groupings of objects grows exponentially with the number of objects, making exhaustive search totally impractical. Rubin and Whitted [51] first attacked this problem through the use of a *structure editor*, an interactive program for constructing successive levels of a hierarchy beginning with the unstructured collection of primitive objects. It allowed the user to look for object coherence in the form of closely clustered objects and to select tight-fitting bounding boxes to enclose them. A means of performing this operation automatically was also suggested in [51]. By viewing the environment as a 3-D histogram and identifying the largest peaks, it should be possible to automate what the human operator was attempting to do by visual inspection.

Weghorst *et al.* [63] suggested that modeling hierarchies used in the construction of an environment are often adequate for the task of ray tracing. The model builder typically groups objects which are in close proximity to one another, and this practice tends to reduce the average projected void area of the resulting bounding volume. Goldsmith and Salmon [23] noted, however, that such hierarchies tend to have large branching factors, thereby reducing the benefits of tree pruning during ray intersection testing. To avoid this problem, they developed a method of automatic generation of bounding volume hierarchies which is closely tied to equation (3), and therefore more appropriate for ray tracing. In their approach, the hierarchy is constructed incrementally, inserting the primitive objects into the growing structure one at a time while striving to minimize the resulting increases in the bounding volume surface areas. Each object is inserted by beginning at the root of the tree and selecting the subtree which would incur the smallest increase in surface area if the new object were to become a child of it. This selection process continues until a leaf of the tree is reached. In the case of a tie at any level, all minimal subtrees are searched, and the one which ultimately produces the smallest increase in the estimated cost of the tree is used.

Goldsmith and Salmon observed that the order in which the objects are inserted into the hierarchy is very important because it can greatly influence the eventual form of the tree. The order imposed by the modeler can be used, but other alternatives include sorting along a line and randomizing. Sorting usually proved to be detrimental, while the best trees were discovered by trying a number of different random shuffles. Since the cost of tree generation is very small compared to the time for ray tracing we can afford to examine many alternatives in the search for an efficient hierarchical organization.

4.4 Approximate Convex Hulls

Convexity is a geometrical property which can often be used to great advantage. It is a particularly desirable property for bounding volumes, for example, because it guarantees that any ray will interesect the volume at most twice, and this is virtually a prerequisite for a simple intersection test. The *convex hull* of an object is a uniquely defined convex volume. It is the intersection of all convex volumes which contain the object, and is therefore the smallest such volume. Together, these facts suggest that convex hulls may be exemplary bounding volumes. However, computing and representing the exact convex hull of an object or collection of objects can be difficult. If we elect to use an approximation of the true convex hull, we can eliminate these problems and, moreover, ensure that the resulting volume will be extremely easy to test for intersection.

The best example of this is a method introduced by Kay and Kajiya [37].

The bounding volumes in their approach are many-sided parallelepipeds which can be made to conform arbitrarily closely to the actual convex hulls of the enclosed objects. The algorithm uses the concept of *plane-sets* which are families of parallel planes. Each plane-set is defined by a single unit vector called the *plane-set normal*, and each plane within a family is uniquely determined by its signed distance from the origin (equal to the inner product of the plane-set normal and any point on the plane). Given a plane-set normal and an arbitrary (bounded) object, there are two unique planes of the family which most closely bracket the object. The infinite region between these planes is called a *slab*, and is conveniently represented by a min–max interval associated with the plane-set normal as shown in *Figure 8(a)*. For polyhedral objects, these values can be computed by forming the dot product of the plane-set normal with each of the object's vertices (in world coordinates), then finding the minimum and maximum of these values. For implicit surfaces such as quadrics the values defining the slab can be computed using the method of Lagrange multipliers [37].

The intersection of several different slabs can define a bounded region enclosing the object, as shown in *Figure 8(b)*. In three-space this requires three slabs whose plane-set normals are linearly independent (two suffice in two-space), but we are by no means limited to three. The greater the number of slabs, the more closely we can approximate the actual convex hull of the object. To intersect a ray with such a volume we first compute the interval along the ray, measured from its origin, which lies within each of the slabs. This amounts to computing two ray–plane intersections for each slab. If the intersection of these intervals is empty, the ray misses the volume. Otherwise, the ray hits the volume and the maximum of the minimum interval values is the distance to the point of intersection.

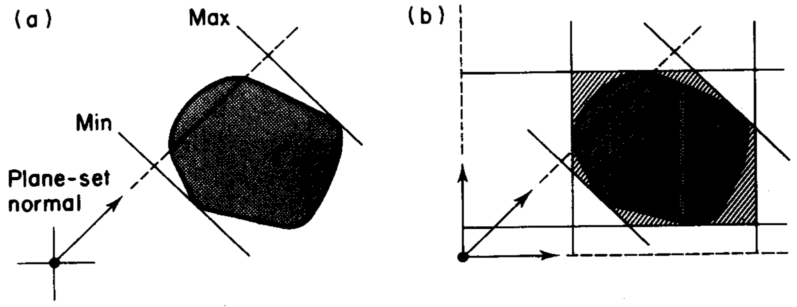

Fig. 8. A plane-set normal defines a family of parallel planes orthogonal to it. Two values associated with a plane-set normal select two of these planes and define a slab. The intersection of several such slabs forms a parallelepiped bounding volume. (a) A single slab bracketing an object. (b) Three slabs defining a bounding volume.

The shape of this type of bounding volume is unaffected by object translation which merely changes the min–max plane constants. Object rotation, on the other hand, affects the quality of approximation. For example, a fairly tight-fitting volume is defined by the three slabs shown in *Figure 6(c)*, but considerably more void area is introduced by a slight rotation of the object relative to the plane-set normals.This suggests that we should select the number and orientation of the plane sets for each individual object in order to obtain very tight-fitting volumes. However, there are tremendous computational advantages to using the same collection of plane-set normals for all the objects of the environment, despite their individual orientations.The most significant advantage is that the task of intersecting a ray with a number of bounding volumes can be greatly accelerated. Common expressions in the ray–plane intersection calculations can be 'factored out' and done once per ray instead of once per bounding volume. When this is done, the calculation requires only two subtracts, two multiplies and a comparison for each slab of a bounding volume [37].

As with other bounding volume techniques, introducing hierarchical nesting is a critical step for efficient execution. To construct a parallelopiped which tightly bounds two or more parallelopipeds (with respect to the same plane-set normals),we compute the new plane constants by finding the minimum and the maximum of all the plane constants associated with each of the plane-set normals. This is directly analogous to constructing nested

```
procedure Hull_Sort_Intersect( in ray, root )
begin
  pre-compute constants for efficient slab intersection;
  initialize heap to contain root bounding volume;
  while heap is not empty do begin
    remove candidate from top of heap;
    if candidate.key > ray.interval.max then return;
    if candidate is a leaf then
      Intersect( ray, candidate.object )
    else for each child of candidate do
      if Intersect_P( ray, child.approx_hull ) then
        add child to heap with key = "estimated distance";
    endwhile
  end
```

Fig. 9. A procedure for intersecting a ray with a collection of objects organized in the hierarchical structure proposed by Kay and Kajiya. In this example 'Intersect_P' makes use of the precomputed constants and also computes the 'estimated distance.'

axis-aligned bounding boxes. In fact, bounding boxes are a special case which results from using the three coordinate axes as the plane-set normals.

Another aspect of the algorithm described in [37] is an efficient method of traversing a bounding volume hierarchy which fully exploits the distance interval optimization described in Section 4. This requires a sorting operation with respect to each ray based on the distances to the points of intersection with the bounding volumes. Kay and Kajiya call these the *estimated distances* because they roughly approximate the distances to the enclosed objects. Using the estimated distance as a key to sort on results in a priority queue for the objects relative to the ray and insures that they are processed in approximately the order that they are encountered by the ray. *Figure 9* is an outline of a procedure which applies this technique to a hierarchy of parallelepipeds formed by intersecting slabs. Here, a *candidate* is any parallelopiped which is intersected by the ray, and each candidate encloses either a single object or a collection of other parallelopipeds. The sorting of the candidates is performed using a heap because, as Kay and Kajiya point out, the efficiency of this operation is quite critical to the performance of the algorithm.

5 3-D SPATIAL SUBDIVISION

The further an object is from the path of a ray, the less work we can afford to do in eliminating it from consideration. As we have seen, bounding volume hierarchies provide a means of recursively narrowing the focus of the search to more promising candidates for intersection. This is a natural divide-and-conquer approach for examining a collection of objects, seeking the member producing the closest intersection. Spatial subdivision begins with a different philosophy. Here we also rely upon simple volumes to identify objects which are good candidates for intersection, but these volumes are constructed by applying a divide-and-conquer technique to the space surrounding the objects instead of considering the objects themselves. Rather than constructing the volumes in a bottom-up fashion by successively enveloping larger collections of objects, we proceed top-down, partitioning a volume bounding the environment into smaller pieces. The smaller volumes thus formed are then assigned collections of objects which are totally or partially contained within them. Therefore a fundamental difference between bounding volume hierarchies and spatial subdivision techniques is that the former *selects volumes based on given sets of objects*, whereas the latter *selects sets of objects based on given volumes*. This leads to a very different approach which places the emphasis on space instead of the objects.

A concept common to all the current techniques of this family is the *voxel*. This is a 'cuboid,' or axis-aligned rectangular prism, and it is the fundamental compartment created by a process of partitioning space. The term itself connotes the extension of 2-D 'picture elements,' or pixels, to 3-D 'volume elements.' A pre-processing step is responsible for constructing nonoverlapping voxels which, taken together, constitute a volume containing the environment. Within these constraints there are different methods of defining the voxels, and these differences lead to the most significant variations within this family. The ramifications of uniform versus nonuniform size are particularly important. Once defined, however, the voxels play the same role in all cases. They are the means of restricting attention to only those objects which are close to the path of a ray.

It the point of intersection between a ray and an object lies within a voxel, both the ray and the object clearly must intersect that voxel. Because the voxels contain the entire environment, every possible point of intersection must lie within some voxel. Therefore, the only objects which we must test for intersection are those which intersect the voxels pierced by the ray. For any given ray, this can potentially eliminate the vast majority of the objects in the environment from consideration. An equally important observation is that a ray imposes a strict ordering on the pierced voxels based on the distance to the point at which the ray first enters each voxel. Because the voxels are nonoverlapping, this ordering guarantees that all intersections occurring within one voxel are closer to the ray origin than those in all subsequent voxels. Consequently, if we process the voxels in the order in which they are encountered along the ray, we needn't consider the contents of any further voxels once we have found a point of intersection (see caveats discussed in Section 5.3). This feature is closely related to the distance interval optimization used in processing bounding volume hierarchies. It can drastically reduce the number of objects which need to be tested and is one of the most attractive features of these techniques.

Spatial subdivision techniques offer an efficient means of identifying the objects which are near the path of a ray while at the same time performing a virtual 'bucket sort' on those objects. In programming terminology, this latter property means that we have moved a portion of the sorting problem from the 'inner loop' of the ray tracing algorithm (as in *Figure 9*) into a pre-processing stage [36]. Naturally, this relies upon the ability to efficiently access the voxels in the order defined by the path of the ray. As we shall see, this operation plays a prominent role in each technique of this family.

5.1 Nonuniform Spatial Subdivision

Nonuniform spatial subdivision techniques are those which discretize space into regions of varying size in order to conform to features of the environment. This variation in size allows more subdivision to be performed in densely populated regions of space and, conversely, it allows large voxels to cover regions which are sparsely populated or entirely void. An *octree* is one possible data structure for creating and organizing such a collection of voxels. Octrees are hierarchical data structures used for efficiently indexing data associated with points in three-space and have been applied to problems such as hidden surface elimination and computation of 3-D digital convex hulls (see [69] and included references). They are constructed by recursively subdividing rectangular volumes into eight subordinate octants until the resulting leaf volumes, or voxels, meet some criterion for simplicity. In most applications the voxels are examined to determine how much of their volume lies within some three-dimensional solid and marked as 'empty,' 'full,' or 'mixed' accordingly.

Glassner [20] introduced an octree variation for use in ray tracing. In this approach each voxel is assigned a list of objects whose surfaces penetrate that volume (*Figure 10*), and these are the intersection candidates for every ray which pierces the voxel. The candidate objects of a given voxel are identified by testing their surfaces against the six faces of the voxel. If a surface intersects one of the faces, the object is immediately added to a candidate list associated with the voxel. For those which do not intersect any of the faces, an additional test for proper containment within this voxel is performed by considering a single point on the object's surface. If the point is inside the voxel, the entire object must be as well (assuming its surface is a connected set), and it is added to the candidate list.

The creation of these candidate lists guides the top-down construction of the octree. A box containing the environment is recursively subdivided until each

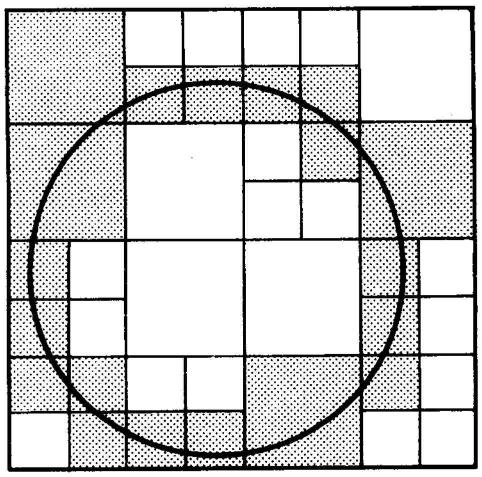

Fig. 10. A 2-D analogy of a sphere and the octree voxels penetrated by its surface. The sphere must be added to the candidate list associated with each of the shaded voxels.

```
procedure Octree_Intersect( in ray )
begin
  Q = ray.origin;
  repeat { walk through the voxels }
    locate the voxel which contains Q;
    for each object associated with voxel do
      Intersect( ray, object );
    if no intersection has been found then
      Q = a point in the next voxel pierced by ray;
  until an intersection is found or Q is outside
    the bounding box of the environment
end
```

Fig. 11. A procedure for intersecting a ray with a collection of objects organized in an octree. These essential features are present in both Glassner's [20] and Kaplan's [35] approach.

voxel contains fewer than some threshold number of intersection candidates or until a storage limitation is reached. Once the octree is constructed, the algorithm outlined in *Figure 11* is used to perform fast ray–environment intersections. Accessing the voxels which are pierced by a ray is accomplished by two fundamental operations within the main loop of this algorithm: (1) 'locating' a voxel which contains a given point in three-space, and (2) constructing a point which is guaranteed to be in the 'next' voxel. The first of these is intimately related to the particular representation of the octree.

In Glassner's approach, nodes of the octree are linked and accessed by uniquely defined *names* rather than storing explicit pointers to descendent nodes. To construct these names a convention of labeling the eight children with the digits 1 ... 8 is adopted. When a node is subdivided the children derive their names by appending their single digit to the name of the parent. Thus, each node of the tree receives a unique integer name consisting of digits which encode the path to that node from the root (which is named '1'). Given the name of a node, the name of any child is obtained by multiplying by 10 and adding the appropriate digit. To access data associated with a node name, such as the candidate list in the case of leaf nodes, the name is used to retrieve a pointer from a hash table. Glassner observed that simply computing the name modulo the size of the hash table serves as a good hashing function. This mechanism is used to retrieve the candidate list associated with a leaf node (voxel) containing a given point. Beginning at the root node we determine which of the eight octants the point lies within, construct the name of the child corresponding to that octant, and consult the hash table to determine the status of that child. The process is repeated until we reach a leaf node or produce the name of a non-existent node, indicating an empty candidate list.

If a ray hits nothing within a voxel we must proceed to the next voxel pierced by that ray. In Glassner's algorithm, this is accomplished by finding a point within the next voxel and performing the look-up described above. To find such a point we first compute where the ray exits the current voxel using a standard ray–box intersection calculation, then move a small distance into the interior of the neighboring voxel, taking care not to step too far. This can be done using the length of the shortest voxel edge in the entire octree; call it 'minlen.' If the exit point is interior to a face of the current voxel, we move the exit point directly away from this face by a distance of minlen/2. If the ray exits through an edge or a vertex of the current voxel we need to make a similar adjustment for each face containing the exit point. The result is a point which lies within the desired voxel.

Kaplan [35] introduced a very similar approach based on binary space partitioning trees (BSP trees), an alternative method for subdividing space into voxels. A BSP tree partitions space into two pieces at each level by means of a separating place. Though Fuchs's hidden surface algorithm [16] employs BSP trees with arbitrarily oriented partitioning planes, Kaplan's approach restricts these to be axis-orthogonal planes and consequently performs nearly the same voxel subdivision as the octree method. One difference between this and Glassner's approach is that the nodes of the BSP tree are constructed with explicit pointers to their two children.This obviates the need for voxel names and hashing at the expense of a potential increase in storage; a typical space/time trade-off.

Jansen [31] introduced a spatial subdivision algorithm based on BSP trees which differs fundamentally from both of the previous methods in the way it identifies the voxels pierced by the ray. Instead of finding the next voxel by creating a point guaranteed to fall within it and traversing the hierarchical structure from the root, we recursively descend all the branches of the BSP tree which terminate at pierced voxels, making use of each partition node only once per ray. Jansen calls the previous method *sequential traversal* and the new method *recursive traversal.* The recursive traversal algorithm is outlined in *Figure 12.* As the BSP structure is traversed, a ray is recursively 'clipped' by each partitioning plane it pierces. That is, the ray's distance interval is divided into two intervals which correspond to segments of the ray on either side of the plane. The segment closest to the ray origin continues the recursive partitioning process first. If this 'near' segment of the ray is found to intersect an object, the 'far' segment is discarded. Otherwise, the far segment of the ray is also recursively partitioned. Frequently the entire ray interval will be entirely to one side of a partitioning plane. When this happens, one of the segments of the ray ('near' or 'far' in *Figure 12*) will be empty, causing the corresponding recursive call to 'BSP_Intersect' to terminate immediately, pruning one of the branches of the BSP tree.

```
procedure BSP_Intersect( in ray, node )
begin
  if ray.interval is empty or node is nil then return;
  if node is a leaf then { this is a "voxel" node }
    for each object associated with the node do
      Intersect( ray, object );
  else begin { this is a "partition" node }
    near = ray clipped to near side of node.partition;
    BSP_Intersect( near, pointer to near half-space);
    if no intersection has been found then begin
      far = ray clipped to far side of node.partition;
      BSP_Intersect( far, pointer to far half-space);
      endif
    endelse
  end
```

Fig. 12. A 'recursive traversal' procedure for intersecting a ray with a collection of objects organized in a BSP tree. Rays are 'clipped' using the distance interval.

Figure 13 shows a 2-D analogy of an environment and the voxels defined by octree subdivision. For simplicity only spheres are depicted here, though the principle is independent of the types of primitive objects used. The subdivision heuristics used to construct this octree were (1) subdivide any voxel with two or more intersection candidates, but (2) subdivide no more than three levels deep. These heuristics are typical of octree approaches, though the values used in this example may not be realistic. Limits placed on the depth

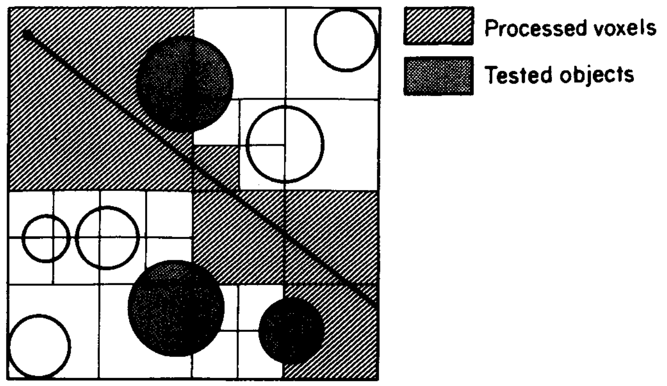

Fig. 13. Non-uniform spatial subdivision via an octree. The ray shown here causes five of the voxels to be examined and three of the eight objects to be tested for intersection. Finer subdivision can decrease the number of ray–object tests at the expense of additional voxel processing overhead.

are usually needed to guard against situations in which the recursion would not terminate based on other criteria. This can occur, for example, when several objects overlap, making them impossible to separate by further subdivision. The processed voxels and tested objects are shaded in this figure for a particular ray. These are independent of the type of traversal algorithm used: sequential or recursive. The ray shown here is a rather bad case, passing through the environment without hitting anything. Nevertheless, only three of the eight objects are tested, and this ratio tends to become more favorable as the complexity of the environment increases. If the ray happens to hit something in a voxel close to the ray origin, fewer voxels need be processed and consequently fewer objects are tested.

5.2 Uniform Spatial Subdivision

Fujimoto *et al.* [17] introduced a different approach to spatial subdivision in which voxels of uniform size are organized in a regular 3-D grid (or lattice). This organization was given the acronym SEADS, for Spatially Enumerated Auxiliary Data Structure. The overall strategy is quite similar to the nonuniform subdivision techniques. Lists of candidate objects are retrieved from voxels which are pierced by a ray and these voxels are processed in the order they are pierced. However, there are two distinguishing features of this approach which are direct consequences of the voxel regularity: (1) the subdivision is totally independent of the structure of the environment, and (2) the voxels pierced by a ray can be accessed very efficiently by incremental calculation. The first is a disadvantage which must be weighed against the obvious benefits of the second. The test cases in [17] are examples where the

```
procedure Grid_Intersect( in ray, node )
begin
  compute i, j, k for the voxel containing ray.origin;
  set up 3DDDA based on ray.direction and ray.origin;
  repeat {walk through the voxels}
    for each object associated with voxel[i,j,k] do
      Intersect( ray, object );
    if no intersection has been found then
      use 3DDDA to compute new i, j, k;
  until an intersection is found or i, j, k is outside
    the limits of the voxel array;
  end
```

Fig. 14. A procedure for intersecting a ray with a collection of objects organized in a uniform grid. The 3DDDA is similar to a line rasterization routine.

speed of voxel access proves to be the dominant factor, indicating that the SEADS approach can sometimes offer significant gains in performance over nonuniform subdivision techniques.

The key to efficient voxel access is that finding the voxels along the path of a ray in a regular lattice is the 3-D analogy of representing a line on a regular array of pixels. To exploit this, Fujimoto *et al.* developed a *three-dimensional digital difference analyzer*, or 3DDDA, to incrementally compute successive voxel indices in the same way that efficient line rasterization algorithms incrementally compute pixel coordinates. One minor difference is that the 3DDDA must step through each voxel which is pierced by the given ray *(Figure 15)*, whereas line rasterization algorithms identify pixels which are merely close to a line in some sense. This requires a departure from the common property of line rasterization algorithms which forces a step to be taken along the dominant axis unconditionally with every iteration.Nevertheless, incremental error terms can still be used to signal discrete steps along the coordinate axes just as in line rasterization. These error terms require careful initialization to correctly handle rays which do not originate from the exact center of a voxel. This is analogous to sub-pixel positioning in line rasterization.

Usually just one of the three indices, i, j, and k, are incremented (or decremented) by the 3DDDA in each iteration of the loop. Exceptions occur when a ray goes through an edge or a corner of a voxel. Because a 3DDDA generates integer coordinate triples, data associated with the voxels is most conveniently stored as a three-dimensional array. In procedure 'Grid_Intersect' (*Figure 14*) this array is named 'voxel' and it is assumed to provide access to the intersection candidates, perhaps by storing a pointer to the head of the list. Given the coordinates of any point interior to a voxel, this

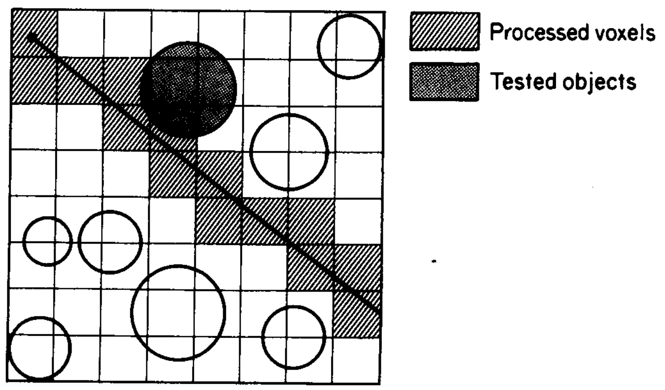

Fig. 15. A 2-D analogy of uniform spatial subdivision. The ray shown here pierces 14 voxels and results in one object being tested for intersection. The uniformity of the grid makes the process of voxel walking similar to representing a line on a rectangular array of pixels.

arrangement allows quick access to its associated data through direct calculation of the voxel indices. This can make the initial construction of the candidate lists quite efficient and is another benefit of uniform subdivision.

The advantages of this approach cannot always compensate for the lack of adaptivity, however. Though the voxels which 'digitize' a ray can be made to approximate the ray with arbitrary precision by increasing the resolution of the grid, two limitations begin to emerge as we do so. First, it becomes more costly to pass rays through empty regions of space, and second, the storage for the corresponding three-dimensional array quickly becomes unmanageable. Of course the storage problem can be alleviated by only storing the voxels which have non-empty candidate lists as Glassner did with octrees [20]. This could be accomplished through a voxel look-up scheme similar to that used by Wyvill *et al.* [67] to construct polygonal approximations of implicitly defined surfaces. This is another space/time trade-off because of the overhead which the hash table look-up adds to the voxel walking process.

Fujimoto *et al.* [17] also made use of the 3DDDA in accessing octree voxels. Because the 3DDDA is applicable only to uniform subdivision, this restricts its use to walking 'horizontally' among sibling voxels of the octree. Each group of eight siblings can be viewed as a small uniform grid and, as such, the 3DDDA provides an efficient means of passing a ray through them. After stepping through at most four voxels, 'vertical' traversal must be performed again in order to locate the next block of eight siblings.

5.3 Two Caveats

There are a number of potential pitfalls which one must be careful to avoid when implementing spatial subdivision techniques. Two in particular stem from the fact that a single object may intersect several voxels, and these pertain to both uniform and nonuniform subdivision techniques. The first is the problem of repeated ray–object intersection tests between the same ray and object. Multiple tests can result from situations such as that depicted in *Figure 16*. As the ray passes through voxels 1 and 2, it finds object A in the candidate list of each. To avoid testing it twice we can employ a *mailbox* as described by Arnaldi *et al.* [2]. A mailbox, in this context, is a means of storing intersection results with individual objects. Each object is assigned a mailbox and each distinct ray is tagged with a unique number. When an object is tested for intersection, the results of the test and the ray tag are stored in the object's mailbox. Before testing every object, the tag stored in its mailbox is compared against that of the current ray. If they match, the object has been previously tested against this ray and the results can be retrieved without being recalculated.

The second caveat is more serious because it can cause erroneous results. A

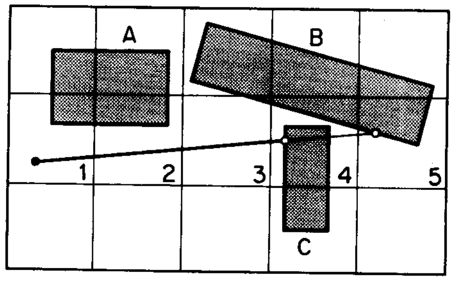

Fig. 16. We can avoid testing object A in voxel 2 after having tested it in voxel 1 by using a mailbox. We must be careful to terminate the voxel walking process only when the point of intersection is contained within the current voxel. Otherwise, object C could be missed in the situation shown here.

situation in which this problem arises is also depicted in *Figure 16.* Notice that object B will be in the candidate list of voxel 3, and this object is indeed intersected by the ray shown. If this affirmative test causes the voxel walking to terminate at voxel 3, the closer intersection with object C will not be found. This can cause disappearing objects. To remedy this we require that the point of intersection be within the 'current' voxel before terminating the search. That is, we only terminate the voxel walking process if the point of intersection is known to be the closest resulting from any object associated with a voxel up to and including the one which contains that point. By using mailboxes we can save the intersection results computed on behalf of an earlier voxel until it is reached by the voxel walking process.

5.4 A Comparison through Graphs

Thus far we have discussed three fundamentally different methods for organizing data in order to accelerate ray intersection calculations: bounding volume hierarchies, uniform spatial subdivision, and nonuniform spatial subdivision. Each involves a relationship between objects and volumes, and each requires a special algorithm for accessing objects based on which volumes are intersected by a given ray. The differences are easy to see if we depict the object–volume organizations required by these algorithms as graphs, as in *Figure 17.* Here, circles represent bounding volumes, squares represent primitive objects, thick lines represent ray intersection tests, and thin lines represent point containment tests (such as used in descending an octree).

Figure 17(a) is the graph resulting from a bounding volume hierarchy. Because there is exactly one path to each of the leaf nodes, the graph is a tree. The children of a node are processed only if the node is intersected by a ray. Different types of bounding volumes and different orders of testing give rise to some of the variations within this family of algorithms. *Figure 17(b)* is the

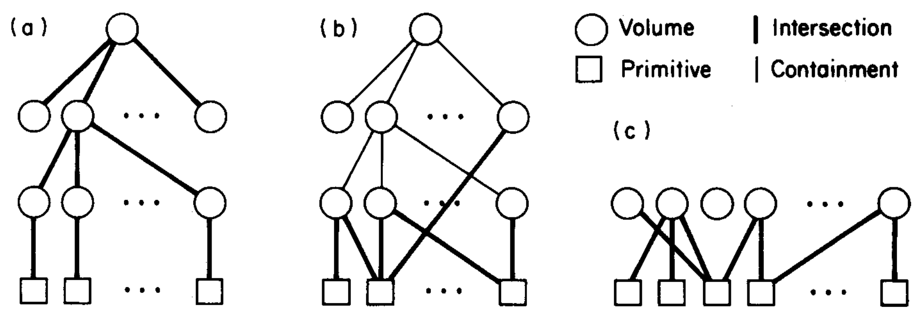

Fig. 17. A bounding volume hierarchy results in (a) a tree, nonuniform spatial subdivision results in (b) a directed acyclic graph, or DAG, and uniform spatial subdivision results in (c) a bipartite graph.

graph resulting from nonuniform spatial subdivision. The subgraph connected by the thin lines is a tree because there is exactly one path to reach any given leaf volume (voxels in this case). These leaf volumes can be associated with any number of objects, and some objects may belong to more than one leaf volume. This makes the overall graph a DAG (directed acyclic graph). Finally, *Figure 17(c)* is the graph resulting from uniform spatial subdivision. Each object belongs to one or more volumes, and each volume contains zero or more objects. There are no explicit edges connecting the volumes because they are accessed by direct index calculation, not by paths through other volumes. Because each edge has one vertex in the set of objects and the other in the set of volumes, the graph is bipartite.

These graphs clearly do not convey all of the features of the corresponding algorithms, but they do serve to highlight the more fundamental differences. Given this representation it is natural to ask which other graphs represent useful algorithms. As a partial answer to this we refer to Section 12 in which combinations of various acceleration techniques are discussed. The corresponding graphs of these hybrid algorithms contain any or all of the ones shown here as subgraphs.

6 DIRECTIONAL TECHNIQUES

The most recent category to emerge is that of directional techniques. Though every ray tracing approach must take ray direction into account, the directional techniques are those which exploit this information at a level above that of individual rays. To see how this differs from other approaches, consider the use of ray direction within a typical 3-D spatial subdivision scheme. Here the direction is used in selecting the subset of voxels pierced by the ray. This eliminates most of the voxels from consideration and defines an efficient order for processing those which remain. However, this selection and ordering of voxels must be performed on a ray-by-ray basis because direction is not taken into account during the construction of the voxels. In contrast, directional techniques explicitly incorporate directional information into data structures which allow more of the overhead to be moved from the 'inner loop' into a less costly stage. Operations such as backface culling and candidate sorting can be done on behalf of many rays instead of individual rays. A common penalty which accompanies these advantages is a very large storage requirement.

There are currently three members in the family of directional techniques: the 'Light Buffer' [25], the 'Ray Coherence' algorithm [47], and 'Ray Classification' [3]. An important mechanism employed by all the members of this family is direction subdivision. Before describing how this is used in each of the algorithms, we introduce some useful terminology and machinery.

6.1 The Direction Cube

A concept which has appeared independently in all three of the algorithms discussed in this section is something which we shall call the *direction cube*. A direction cube plays a similar role to that of the 'hemi-cube' used in the radiosity method [8]. It is a means of discretizing directions into a finite number of square or rectangular *direction cells* and is analogous to spatial subdivision methods which discretize bounded regions of space into a finite number of voxels. More precisely, a direction cube is an axis-aligned cube centered at the world coordinate origin. The six faces of this cube correspond to six *dominant axes* which we label $+X$, $-X$, $+Y$, $-Y$, $+Z$ and $-Z$ (see *Figure 18(a)*). Each of these faces subtends a solid angle of $2\pi/3$ steradians from a vantage point of the coordinate origin.

The direction cube allows us to translate 3-D directions into the language of 2-D rectangular coordinates. To account for 4π steradians (i.e. all possible directions) we define these 2-D rectangular coordinates, designated u and v, on six independent squares corresponding to the faces of the direction cube. For any given ray, we can then construct an alternative representation for its direction by imagining it translated to the coordinate origin, determining which face of the direction cube it intersects (i.e. finding the dominant axis of the ray) and then computing the u–v coordinates of the point of intersection (*Figure 18(a)*). Scaling the U and V axes so that the cube edges are of length two guarantees that all points of intersection will have coordinates between -1 and 1. This convention makes calculations particularly efficient. *Figure 19* shows a procedure for performing this mapping by defining U and V axes on each face as synonyms for two of the world coordinate axes, X, Y, or Z. The exact correspondence chosen in each case is immaterial, so this procedure

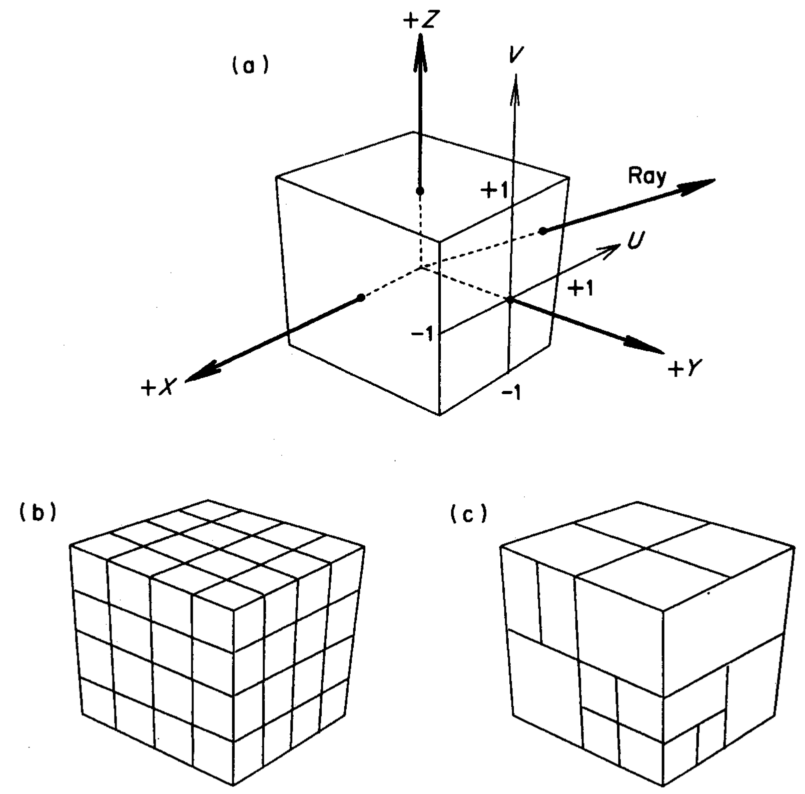

Fig. 18. The direction cube is used to translate 3-D directions 2-D rectangular coordinates. This provides a means of applying subdivision techniques in the context of directions. (a) Three of the dominant axes associated with a direction cube. (b) Uniform subdivision. (c) Adaptive subdivision.

simply selects U and V to be the two axes, in lexicographic order, which are parallel to the face.

This translation to 2-D rectangular coordinates allows us to easily and efficiently apply subdivision techniques in the context of directions. Just as in spatial subdivision, we can choose to subdivide the squares uniformly or nonuniformly, and in the latter case, standard techniques such as BSP trees or quadtrees are applicable. Examples of uniform and nonuniform direction subdivision are depicted in *Figures 18(b)* and *18(c)*, respectively. Each direction cell resulting from subdivision defines an infinite skewed pyramid with its apex at the coordinate origin and its edges through the cell corners. We shall refer to these as *direction pyramids*. A direction pyramid is the volume of space accessible to rays which begin at the coordinate origin and pass through the given direction cell. Notice that even in the case of uniform subdivision the direction cells do not all subtend equal solid angles. This poses no problem, however, because it is the efficiency of the translation from directions to 2-D coordinates which is important to the algorithms in the following sections, not the exact shapes of the direction pyramids.

```
procedure Direction_to_UV( in direction; out axis, u, v )
begin
   ax = | direction.x |
   ay = | direction.y |
   az = | direction.z |
   if ax > ay and ax > az then begin { X is dominant }
      if direction.x > 0 then axis = pos_X else axis = neg_X
      u = direction.y / ax
      v = direction.z / ax
      endif
   else if ay > az then begin { Y is dominant }
      if direction.y > 0 then axis = pos_Y else axis = neg_Y
      u = direction.x / ay
      v = direction.z / ay
      endif
   else begin { Z is dominant }
      if direction.z > 0 then axis = pos_Z else axis = neg_Z
      u = direction.x / az
      v = direction.y / az
      endelse
   end
```

Fig. 19. A procedure for mapping 3-D direction vectors into points in one of six 2-D rectangular coordinate systems indexed by the six dominant axes.

Given a subdivided direction cube, it is a simple matter to determine which direction cell is pierced by any ray. We begin by determining the ray's dominant axis and $u-v$ coordinates, as discussed above. Then, in the case of uniform subdivision, the row and column indices of the direction cell containing this 2-D point are found by direct calculation. In the case of nonuniform subdivision more work is required, such as traversing a hierarchical partitioning structure for that face. These cases are the exact analogs of problems encountered in uniform and nonuniform spatial subdivision techniques. Furthermore, the role played by the direction cells is similar to that of voxels. Both are used to access lists of candidate objects, indexed by direction neighborhoods in one case and by spatial neighborhoods in the other. In both cases the purpose of locating the appropriate neighborhood is to retrieve the associated candidate list.

We now turn to applications of this directional information. The key to understanding the connection between the three algorithms in this section is to observe that they each begin by associating direction cubes with specific collections of rays. The most straightforward application is to consider only rays which originate from a finite number of isolated points. Special points which are particularly appropriate are point light sources (which we can think of as emitting the rays used in shadow testing) and the eye point. The 'Light Buffer' algorithm [25] was developed to take advantage of the former case. Other collections of rays which can be associated with direction cubes are those which originate from the surfaces of individual objects. The 'Ray Coherence' algorithm [47] takes this approach. Finally, we can associate direction cubes with collections of rays which originate from 3-D voxels. The 'Ray Classification' [3] algorithm is closely related to this concept, although as we shall see it partially removes the distinction between direction cells and voxels.

6.2 The Light Buffer

The Light Buffer, introduced by Haines and Greenberg [25], is a directional technique which accelerates the calculation of shadows with respect to point light sources. One of the facts exploited by this algorithm is that points can be determined to be in shadow without finding the closest occluding object. Since any opaque occluding object will suffice, shadowing operations are inherently easier that normal ray–environment intersections. Furthermore, constraining light sources to be single points allows a particularly effective application of the direction cube to these operations.

The search for an occluding object can be narrowed to a small set of objects by making use of the direction from the light source to the point in question. The light buffer algorithm accomplishes this by associating a uniformly subdivided direction cube with each light source, and a *complete* list of candidate objects with each of the direction cells. That is, each candidate list contains every object which can be 'seen' through the corresponding direction cell. These candidate lists are retrieved by finding the direction cell pierced by each light ray which is (conceptually) cast from the light source. The objects in this list are the only ones which can block the ray and thereby cast a shadow.

The light buffers are constructed as a pre-processing step, before ray tracing begins. The candidate lists are created by projecting each object of the environment onto the six faces of each direction cube, adding them to the candidate lists of those direction cells which are partially or totally covered by the projection. For polygonal objects this is performed efficiently by applying a modified scan-line algorithm to the projected edges. Nonpolygonal objects can be enclosed in polyhedral hulls for the purpose of creating the candidate lists, although the actual shadow intersection testing must use the object itself. Once all the lists are created, they are sorted into ascending order according to depth.

There are several observations which can lead to simplified candidate lists.

First, all polygons which face away from the light and are part of opaque solids may be culled. Also, any list which consists of exactly one polygon can be deleted, because a polygon cannot occlude itself unless it is facing away from the light. Finally, if the projection of an object completely covers a direction cell, the candidate list can be terminated by a *full-occlusion record* at the object depth, and all candidates at a greater depth can be eliminated. The direction pyramid from this depth onward is completely in shadow with respect to that light source. In order to exploit this optimization for objects with curved surfaces, we can detect totally covered direction cells by using enclosed polygonal meshes instead of bounding hulls.

To determine if a point on a given surface is in shadow, we first check the orientation of the surface with respect to the light source. If it is facing away, the polygon is known to be in shadow. Otherwise, we retrieve the list of potential occluding objects from the light buffer using the direction of the light ray. The objects in the list are then tested for intersection, in order of increasing depth, until an occlusion is found or until we reach an object whose depth is beyond the point we are testing. In the former case the point is in shadow, and in the latter case it is illuminated. If the list is marked with a full occlusion record and the point we are testing is at a greater depth, we can immediately conclude that the point is in shadow without performing any intersection tests. Note that this optimization is one of the benefits of the special treatment of light rays. It is irrelevant whether the object causing full occlusion is the first one hit by the light ray.

6.3 The Ray Coherence Algorithm

In this and the following section we describe algorithms which extend the use of directional information to the acceleration of general intersection calculations. Ohta and Maekawa [47] achieved this through application of what they have termed the 'ray coherence theorem.' This is a mathematical tool for placing a bound on the directions of rays which originate at one object and then hit another, making it possible to broaden the application of direction cubes from single points to bounded objects. In its simplest form, the ray coherence theorem applies to objects which are bounded by nonintersecting spheres, as in *Figure 20*.

Any ray which originates within sphere S_1 and terminates within sphere S_2 defines an acute angle, θ, with the line through the sphere centers. Inequality (4) is a bound on this angle in terms of the sphere radii and the distance between their centers.

$$\cos\theta > \sqrt{\left(1 - \frac{r_1 + r_2}{\|O_1 - O_2\|}\right)}. \tag{4}$$

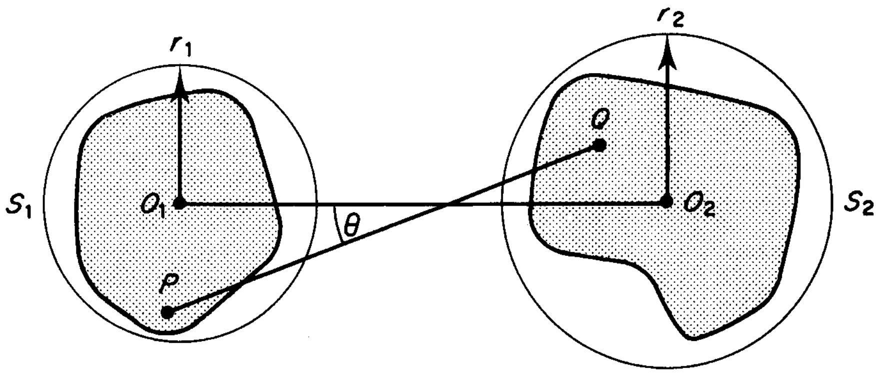

Fig. 20. A bound on the angle between lines O_1O_2 and PQ can be computed in terms of the distance between the centers of the two bounding spheres and their radii. This can be used to bound the directions of the rays which originate at one object and intersect the other.

Ohta and Maekawa also used a version of this theorem which is applicable when the objects are bounded by convex polyhedra instead of spheres. The resulting direction bound is phrased in terms of a 2-D convex hull on the surface of a sphere. In either case, approximations of these direction bounds are stored by means of uniformly subdivided direction cubes associated with each entity in the environment from which rays can originate. This includes the eye point, light sources, and reflective or refractive objects. Each of these direction cubes is constructed and used in nearly the same manner as a light buffer. A pre-processing operation creates depth-sorted lists of intersection candidates for each direction cell of each direction cube. These candidate lists determine the objects which need to be tested for intersection with any ray based on its direction and the entity whence it originated. The direction cube therefore accelerates the process of finding the 'next' object hit, providing an efficient way of progressing from object to object as the path of a ray is traced. This essentially reduces to a light buffer in the case of shadow tests with respect to point light sources.

Departures from the light buffer algorithm occur in both the construction and intersection testing of the candidate lists. During candidate list construction, objects are associated with individual direction cells by means of a relationship such as (4) rather than by projecting the objects (or their bounding volumes) on to a single point. This is equivalent to sweeping the center of projection over the object from which the rays originate and identifying all the direction cells which are touched by the projections. The difference in testing a candidate list for intersection is that nonshadowing rays require that the closest point of intersection be found. Objects in the list are tested *in order* until the list is exhausted or the minimum distance, given by d in

```
for each entity, A, from which rays can originate do begin
  for each object B do begin
    d = lower bound on distance between A and B;
    S = direction bound for rays from A which hit B;
    for each direction cell of A which intersects S do begin
      insert B into the cell's sorted candidate list
      according to the distance d;
      endfor
    endfor
  endfor
```

Fig. 21. The pre-processing algorithm of the 'ray coherence' algorithm. The sorting operation can be performed by insertion, as shown here, or after all the candidate lists have been formed.

Figure 21, is greater than the distance to a known point of intersection. This is the distance interval optimization yet again.

An outline of the pre-processing algorithm which creates the candidate lists is shown in *Figure 21*. It is assumed that bounding volumes are all spheres or all convex polyhedra. The direction bound, S, will be a unit vector and an angle (or cosine) defining a cone in the case of bounding spheres and a spherical convex hull in the case of bounding polyhedra.

6.4 Ray Classification

The ray classification algorithm, described by Arvo and Kirk [3], does not use explicit direction cubes except in the special case of first-generation rays. The data structure used to accelerate the intersection process for other rays is closely tied to the concept of a direction cube, however. Ray classification is based upon the observation that rays in three-space have five degrees of freedom and correspond to the points of $R^3 \times S^2$, where S^2 is the unit sphere in three-space. The algorithm proceeds by partitioning the five-dimensional space of rays into small neighborhoods, encoded as 5-D hypercubes, and associating a complete list of candidate objects with each. A hypercube represents a collection of rays with similar origins and similar directions, and its associated candidate list contains all objects which are hit by any of these rays (neglecting occlusion). To intersect a ray with the environment, we locate the hypercube which contains the 5-D equivalent of the ray and test only the objects in the associated candidate list.

Rays with a given dominant direction can be conveniently encoded as 5-tuples, (x, y, z, u, v), where the first three elements specify the origin of the ray, and the last two are the UV direction coordinates obtained from a face of the direction cube. Any ray in three-space can be specified by such a 5-tuple and an element of the set $\{+X, -X, +Y, -Y, +Z, -Z\}$. If B is a 3-D box which contains the environment, then a set containing all rays which are relevant to this environment can be represented by six 'copies' of the space $B \times [-1, 1] \times [-1, 1]$. These ***bounding hypercubes***, corresponding to the six dominant axes, are a basis for combined spatial and directional subdivision using a hyper-octree, a 5-D analog of an octree. The 5-D hypercubes at the leaves of the hyper-octree are assigned lists of candidate objects in direct analogy with the voxels of a 3-D spatial subdivision scheme. We find the candidate list for a given ray by converting the ray into a 5-tuple and, beginning at the root of the hyper-octree corresponding to the ray's dominant axis, traversing the tree until we find the leaf node containing that 5-tuple. The most recently accessed hypercubes can be cached in order to avoid this hierarchy traversal in most cases.

To construct the candidate lists, we observe that a 5-D hypercube represents a collection of rays which originate from a 3-D voxel and possess directions given by a single direction cell. This collection of rays sweeps out an unbounded 3-D polyhedral volume called a ***beam***. See ***Figure 22***. The candidate list of a hypercube must contain all objects which intersect this beam. As the nodes of the hyper-octree are subdivided, a child's candidate list can be obtained from the parent list by removing those objects which fall outside of its narrower beam. By bounding objects with convex polyhedra, the operation of comparing objects with a beam reduces to detecting polyhedral

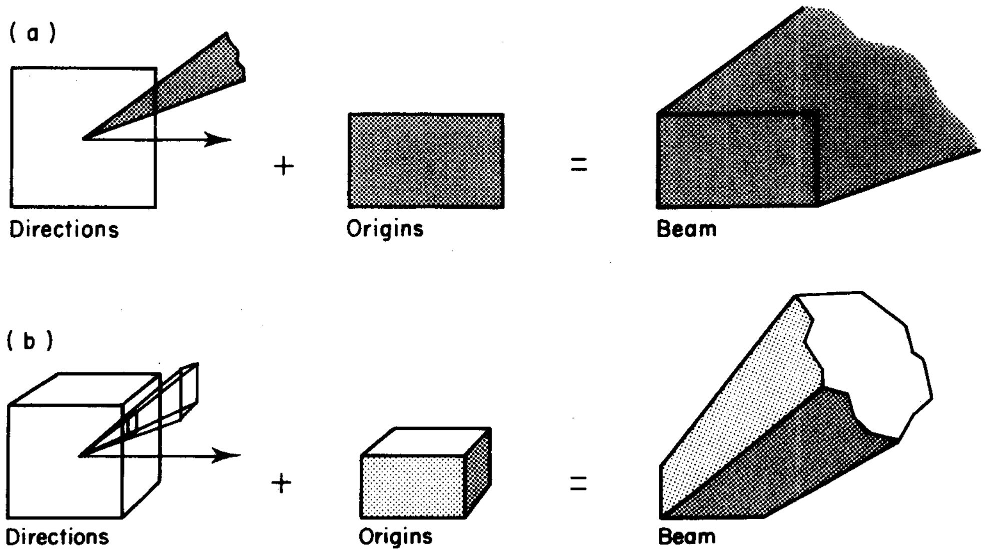

Fig. 22. Beams in (a) 2-space and in (b) 3-space. A beam can be defined as the set sum of the direction pyramid and the voxel. That is, every point of the beam can be expressed as the vector sum of a point within the pyramid volume and a point within the voxel.

intersections. This can be solved by linear programming, for example, however Arvo and Kirk found this to be too costly. An effective alternative is to bound objects by spheres and beams by cones. The cone–sphere intersection test is only a rough approximation, but it is very fast. It is also possible to use object hierarchies for efficient creation of candidate lists. The techniques described by Dadoun and Kirkpatrick [13] for the acceleration of 'beam tracing' may be useful here.

As with the other directional techniques, the candidate lists are sorted by depth in order to most effectively apply the distance interval optimization. A difference in the ray classification approach is that only the candidate lists associated with the original bounding hypercubes need be sorted. These lists contain all the objects in the environment and the sorting is with respect to minimum object extents along the six dominant directions. All subsequent lists are derived from these by deleting entries, so the sorted order can be passed down with no additional work.

Because the hyper-octrees and associated candidate lists can potentially become very large, there are a number of important space-saving measures which can be applied. By far the most critical of these is restricting subdivision to occur only in regions of 5-space which are actually populated by rays of interest. The best way of achieving this is to subdivide only on demand, as rays are being traced. Building the entire data structure by lazy evaluation saves a vast amount of storage because the rays which are actually used occupy a very sparse subset of the rays represented by the bounding hypercubes.

Another means of saving space is to store only partial candidate lists. We can truncate a candidate list at a given distance from the beam origin and discard all of the objects which lie entirely beyond this distance. In order to complete the tracing of a ray which is not intercepted by any of the remaining objects, we 'push' the ray origin up to the truncation plane and begin anew with this ray. Carried to an extreme, this discarding of information makes the ray classification algorithm resemble non-uniform 3-D spatial subdivision with sequential traversal (see Section 5.1).

Figure 23 shows the organization of the ray classification algorithm. The notation C(H) means the candidate list associated with hypercube H, and C(H) $\cap$ Beam(H) means the subset of this candidate list which intersects the beam defined by H. Another aspect of lazy evaluation is that we do not form the candidate list of any hypercube until it is actually needed. When we subdivide a hypercube into 32 children (by splitting along each of 5 axes), only one of the children receives a newly created candidate list. The others simply *inherit* (a pointer to) the parent's list. Only when these other children are visited by a ray will their candidate lists be intersected with their beams.

```
                    procedure RC_Intersect( ray )
                    begin
classification   {  p = the 5-tuple corresponding to ray;
of the ray       {  axis = dominant axis of ray;
                 {  H = the leaf hypercube of hyper-octree[ axis ] containing p;
lazy             {  if C(H) is "inherited" then C(H) = C(H) ∩ Beam(H);
subdivision      {  while C(H) is too large and H is not too small do begin
& candidate      {      partition H along each of the 5 axes;
list creation    {      Let all the new children "inherit" C(H);
                 {      H = the child hypercube which contains p;
                 {      C(H) = C(H) ∩ Beam(H); { reclassify candidates }
                 {      endwhile;
candidate        {  for each candidate in C(H) do begin { stepping in
processing       {      ascending order }
                 {      d = projection of ray.interval.max onto axis;
                 {      if d < candidate.min then return; { past distance interval }
                 {      Intersect( ray, candidate );
                 {      endfor
                 {  end
```

Fig. 23. An outline of the ray classification algorithm. The construction of the 5-D hierarchy is an integral part of the algorithm because it occurs as a side effect of tracing rays. Subdivision continues until the candidate list is sufficiently small, or *H* becomes too small.

6.5 Comparing the Directional Techniques

Figure 24 shows a number of important similarities and differences among the directional techniques. Only features in which there is some variation are shown. Most of the differences are a direct result of nonuniform versus uniform direction subdivision. For instance, nonuniform subdivision leads naturally toward lazy evaluation and also requires more parameters and heuristics to control it. Conversely, uniform subdivision requires few parameters and leads to very efficient look-up, but also encourages construction as a pre-processing step. Note that these are not necessarily immutable properties but merely a reflection of the initial descriptions of these algorithms. Improvements and generalizations are no doubt possible in each case.

	Light buffer	Ray coherence	Ray classification
ections ssed with	Points (representing light sources)	Objects (including light sources)	Space (bounding environment)
plies to	shadowing rays	all rays	all rays
When data structure is built	preprocessing	preprocessing	lazily during ray tracing
Construction of candidate list	modified scan-line algorithm	'coherence theorem' applied to pairs of objects	object classification using hierarchy of beams
Direction subdivision	uniform	uniform	nonuniform
Parameters	direction cube resolution	direction cube resolution	max tree depth, max candidates, truncation size, etc.
Candidate list look-up	direct calculation given ray direction and light source	direct calculation given ray direction and object of origin	traversal of 2-D or 5-D hierarchy and caching

Fig. 24. Comparisons within the family of directional techniques.

7 EXPLOITING COHERENCE

Why is it that we can expect to design algorithms which perform better than exhaustive ray tracing? The answer lies in properties of the environment which are often tacitly assumed. These are properties which insure that the environment is well behaved in some sense, and are usually expressed in terms of some form of *coherence*.

Sutherland *et al.* [57] identified many types of coherence which can be exploited by hidden surface algorithms. There are four types which are commonly exploited in the context of ray tracing. Of these, *object coherence* is the most fundamental. It expresses the fact that objects tend to consist of pieces which are connected, smooth, and bounded, and that distinct objects tend to be largely disjoint in space. Objects are not typically intermingled clouds of randomly scattered fragments. *Image* (or *scene*) *coherence* is the view-dependent version of object coherence. It expresses the fact that object coherence carries over to 2-D projections of the environment. That is, we have at least the same degree of connectedness, smoothness, etc. in the image plane as existed among the original 3-D objects. *Ray coherence* means that similar rays are likely to intersect the same object in the environment. Thus, two rays which have nearly the same origin and nearly the same direction are likely to trace out similar paths through the environment, hitting the same objects in nearly the same places. This is clearly related to connectedness and smoothness properties of the objects, and is therefore another manifestation of object coherence. *Frame coherence* is essentially image coherence with an added temporal dimension. It means that the projection of an environment tends to change continuously over time. In other words, two successive 'frames' of an animation are likely to be similar if the difference in time is small. This again depends upon object coherence, but with the added property that objects (including the eye and light sources) tend not to move chaotically with time.

Spatial subdivision techniques rely heavily upon coherence, though this dependence is rarely stated explicitly. The fact that small voxels tend to intersect relatively few objects in the environment (i.e. that objects tend to be 'locally separable') is directly attributable to object coherence. This property is precisely what makes spatial subdivision work. If the candidate lists associated with voxels could not be made significantly simpler (on average) than the original environment, spatial subdivision would gain nothing over exhaustive ray tracing. In addition, other aspects of object coherence tend to lessen the impact of 'difficult' voxels.If the objects associated with a voxel are not separable by further subdivision, they will tend to intercept most rays which pierce the voxel. As a result, the penalty of large candidate lists is at least partially counterbalanced by a greater likelihood of terminating the voxel walking. Kaplan [36] observed that this compensating effect can keep the cost of ray tracing relatively insensitive to the number of objects in the environment. Though the complexity of individual voxels may increase, fewer voxels are processed per ray on average.

Ray coherence is more difficult to exploit directly than object coherence, though several approaches do so successfully. Among these are the generalized ray techniques which will be described in Section 9. These rely upon the fact that bundles of similar rays interact with the environment in a fairly uniform way, making it significantly more efficient to process them as a group than individually. As with individual rays, we can expect a narrow cone or beam to miss most of the objects in the environment. This fact allows much of the work involved in ray–environment intersection testing to be shared among many rays.

Perhaps the most direct use of ray coherence in the setting of standard ray tracing was attempted by Speer *et al.* [56]. In this approach the entire ray tree resulting from a first-generation ray is retained in order to serve as a guide for the construction of one or more subsequent ray trees. Ray coherence implies that similar first-generation rays are likely to produce similar ray trees. The problem addressed by Speer *et al.* was that of quickly identifying situations in which a ray tree will have exactly the same structure as the previous one, intersecting the same objects in the same order. If this were known *a priori*, the new tree could be constructed very efficiently from the old one, performing exactly one ray–object intersection calculation for each ray.

In the absence of such *a priori* knowledge, Speer's approach examines each ray of the new tree to determine which of them 'behaves coherently.' That is, to identify the rays which (1) hit the same object as the corresponding ray of the previous tree, and (2) do not hit any new intervening objects. The first can be verified by a direct ray–object intersection calculation. In order to quickly verify the second, each ray of the retained tree is given a cylindrical *safety zone* which is as large as possible without intersecting any objects aside from those at which the ray originates and terminates. *Figure 25(a)* shows the safety zones for a ray tree consisting of two rays. If the corresponding rays of the next tree intersect the same objects and do not pierce any of these cylinders, then no other objects need be checked. This is the case of the dashed ray in *Figure 25(a)*. If any cylinder is pierced, a more costly method is needed to find the appropriate point of intersection, and the retained ray tree must be updated with new objects and safety zones. Test results reported in [56] indicated that a large percentage of the rays can be handled in a 'coherent' manner. Unfortunately, the cost of testing and maintaining the cylindrical safety zones were found to negate the benefits of this coherence.

Hanrahan [29] achieved better success with a related method. This method also retains an entire ray tree but differs from Speer's approach in that it does not attempt to guarantee unobstructed passage from one object to the next. Instead, all objects which can possibly prevent a ray from reaching the previously hit object are identified using cones circumscribed around pairs of objects (*Figure25(b)*) and are associated with the retained ray tree. This retained tree is used as a cache, indicating which objects are likely to be hit by each ray of a new ray tree, and also providing enough information to determine when the 'hint' fails. A cache miss occurs when the ray either misses the previously hit object or hits one of the potential blockers. When a cache miss occurs, the tree is updated and new potentially blocking objects are identified. Though the number of potential blockers may be large, requiring an equal number of ray–object intersection tests, a greater number of coherent rays are tracked and no ray–cylinder intersection checks are needed.

The directional techniques of Section 6 all exploit ray coherence in a natural way. Each algorithm constructs candidate lists which are associated with neighborhoods of similar rays, though these neighborhoods are defined

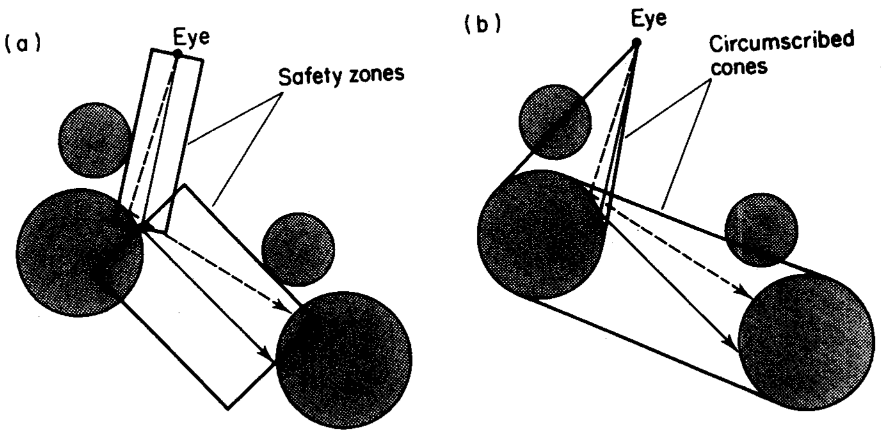

Fig. 25. Two methods of using a previous ray tree to accelerate subsequent intersection tests. In (a) cylindrical safety zones are used to determine when a new object may be hit. In (b), objects which intersect the circumscribed cones are potential blockers which cause cache misses.

differently in each case. One such collection of candidates can be efficiently shared among all the rays of a neighborhood by virtue of the fact that similar rays tend to interact with the environment similarly.

8 STATISTICAL OPTIMIZATIONS

Statistical methods have begun to play an important role in ray tracing. Cook *et al.* [10] described a stochastic sampling technique which provided a means of anti-aliasing as well as simulating effects such as motion blur, penumbrae, depth of field, and dull reflections. Kajiya's rendering equation [34] extended these ideas to simulate effects such as caustics and diffuse interreflection of light between objects. In most implementations, the color to be displayed at a single pixel can be viewed as a weighted integral of the *image function* over a neighborhood of the pixel, where the weight may be a filter for anti-aliasing [42]. Stochastic sampling serves to compute reliable estimates of these integrals via Monte Carlo integration. Naturally there is always a degree of uncertainty in such estimates, although we can produce results of arbitrarily high precision by computing the mean of a large number of samples.

To reduce the expense we wish to draw only enough samples to produce an estimate of the desired accuracy. For example, we may wish to draw the minimum number of samples which will place the estimate within 1% of the true solution with 95% confidence. For maximal efficiency we need to establish a relationship between the number of samples and the quality of the estimate without resorting to rules of thumb such as '*n* is usually enough.' If we knew the variance of the image function over each pixel *a priori*, we could pre-compute the appropriate number of samples which need to be drawn from each. Unfortunately, this type of information is very hard to produce, especially when the number of dimensions being sampled is large due to effects such as motion blur and depth of field [9]. A more practical solution is to rely upon the samples which are drawn not only to estimate the integral of the image function, but also the variance of the estimator. If the variation among the initial samples is sufficiently small, no further samples need to be drawn.

After obtaining each sample we can compute a new estimate of the true variance over the pixel. Such an estimator is itself a random variable, and its distribution is related to the chi-square distribution if we assume that the original samples are normally distributed. Lee *et al.* [42] used this fact to devise a convenient stopping criterion for the stochastic sampling process. Two parameters, T and β are used to control the quality of the image. The tolerance, T, specifies the acceptable variance of the computed pixel values, and β is the probability of stopping too early. That is, β is the probability of incorrectly inferring that the true variance is sufficiently low that the samples drawn thus far will provide a good estimate. The parameters T and β determine threshold values which are pre-computed and stored in a table. When the Nth sample is drawn, the estimated variance is incrementally updated and compared with the Nth entry in the table. If the computed value is less than the table entry, the sampling stops and the mean of the N samples is used as the pixel value. A very similar approach based on the Student t-test was described by Purgathofer [48].

9 GENERALIZED RAYS

The difficulty of anti-aliasing and exploiting coherence in ray tracing stems from its use of infinitesimally thin rays. Though the simple form of these rays leads to easy representation, efficient intersection calculations, and great generality, some of these benefits can be traded in exchange for others. One way to do this is to dispense with individual rays and, instead, operate simultaneously on entire families of rays which are bundled as beams [30], cones [1], or pencils [54]. Each of these *generalized rays* requires some type of sacrifice. For instance, we may need to impose constraints on the environment, such as restricting the types of primitive objects, or we may need to abandon the notion of 'exact' intersection calculations, accepting an approximation instead. The advantages gained in return can include faster execution, effective anti-aliasing, and even additional optical effects.

Amanatides [1] generalized rays to right circular cones which are represented by an apex, center line, and spread angle. For the purpose of anti-aliasing, the intersection calculation not only needs to detect when a cone and an object intersect, but how much of the cone is blocked by the object. A sorted list of the closest few objects which intersect the cone is required so that the partial coverages can be properly combined. For reflection and refraction, the new center line is computed using standard ray tracing techniques. The calculation of the new virtual origin and spread angle required knowledge of the surface curvature. The method of cone tracing also extends the repertoire of ray tracing to include penumbrae (from area light sources) and dull reflections. Due to the difficulty of the cone intersection and partial coverage calculation for most objects, the environment is restricted to spheres, planes, and polygons.

Kirk [38] extended the cone technique by accelerating the processing of partial intersections. The projected area of cone–sphere and cone–plane intersections can be pre-calculated for a wide range of cases and stored in a table. Using a table look-up instead of a direct calculation produces an approximate but fast partial coverage calculation. The cone area at the intersection can also be used to properly anti-alias procedural textures. The cone radius at the intersection determines the aperture size of the smallest feature which should be represented in the texture.

Heckbert and Hanrahan [30] introduced a different ray generalization in their *beam tracing* algorithm. In this approach rays are replaced by beams which are cones with arbitrary polygonal cross section. That is, a beam consists of a collection of rays which originate at a common apex and pass through some planar polygon. Note that this is different from the definition of a beam in the context of the ray classification algorithm discussed in Section 6.4. There the rays are restricted to pass through a rectangular polygon and the origins are not restricted to a single point.

The restriction placed on the environment by this algorithm is that all objects must be constructed with planar polygonal facets. This preserves the basic characteristics of beams under various interactions with the environment. For instance, the portion of a beam which continues past a partially occluding object still has polygonal cross section (*Figure 26*), as do beams which are reflected from any surface (*Figure 27*). Refraction is the one phenomenon which does not preserve the nature of beams. Because of nonlinearity, a refracted beam may no longer be a cone. One remedy is to approximate the effect of refraction with a linear transformation. This is another compromise which must be made in order to obtain the benefits of beam tracing.

Many aspects of the beam tracing algorithm are very similar to those of standard ray tracing. A *beam tree* is constructed by recursive reflection and transmission of beams, though the process of applying these operations to

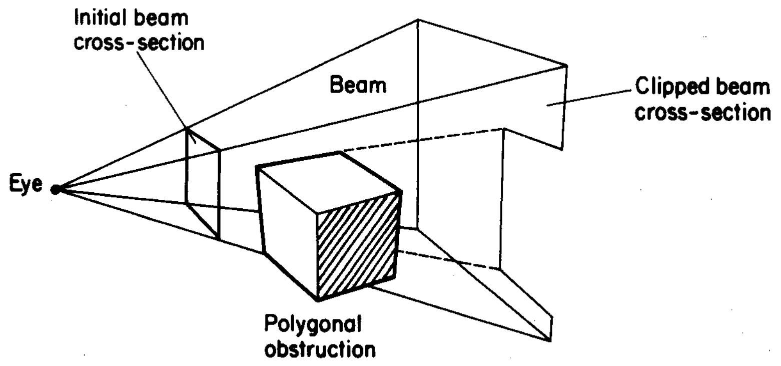

Fig. 26. A polygonal obstruction is clipped out of the cross section of a beam. This operation can quickly lead to cross sections which are non-simple polygons (e.g. disconnected with holes).

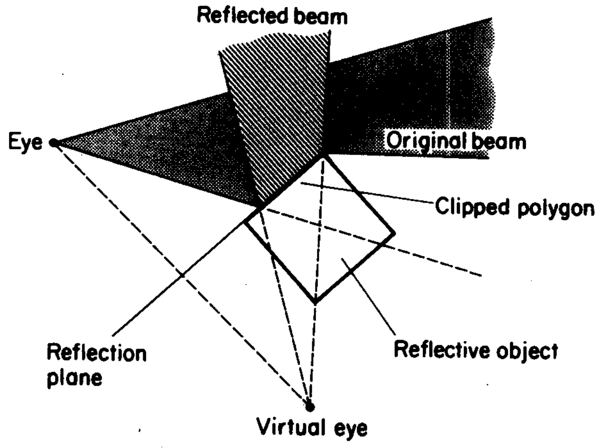

Fig. 27. A top view of the arrangement in Fig. 26. When a reflective face is encountered by a beam, the reflected beam is formed by reflecting the original apex through the plane of the polygon, and clipping the polygon against the beam.

beams is more complex than the corresponding operations used in constructing a standard ray tree. When reflective surfaces are encountered, a 'virtual eye' point is computed by reflecting the apex of the beam through the plane of the polygon. The reflected beam has the virtual eye as its apex and its cross section is obtained by (effectively) intersecting the reflective polygon with the beam. See *Figure 27.* An important property of beams is that they can be partially occluded, whereas rays either hit an object or not. When a beam is partially occluded we 'clip out' the silhouette of the obstruction from the beam cross section and continue processing the remainder. See *Figure 26.* This clipping of the beam is a generalization of the distance interval optimization for rays. It makes it possible to avoid processing far away objects which are occluded by near ones. Heckbert and Hanrahan [30] therefore performed a sorting operation on the polygons intersected by the beam before processing them.

The beam clipping at the heart of the beam tracer requires operations on polygons similar to those described by Weiler and Atherton [64]. We must be able to subtract one polygon from another, or find their intersection, and express the result as another polygon. These operations can quickly lead to nonconvex or fragmented polygons containing holes. Because of the recursive nature of the beam tracing algorithm, the output of one such operation may become the input to another. This requires robust methods which can operate on arbitrarily complex polygons.

Beam tracing can be broken down into three basic subproblems: intersection, sorting, and clipping. Dadoun and Kirkpatrick [13] showed that all three of these can be accelerated by introducing a *hierarchical scene representation*. This data structure employs both nested convex hulls and partitioning planes, combining aspects of bounding volume hierarchies with spatial subdivision. To construct it, the environment is first recursively subdivided, top-down, using a BSP tree. As in Fuch's hidden surface algorithm [16], the partitioning planes may be selected to contain given polygons in the environment. All remaining polygons are grouped according to the two half-spaces defined by the plane, which requires splitting polygons which straddle the plane. After the BSP decomposition, we build a binary tree of nested convex hulls, bottom-up, beginning at the leaves of the BSP tree. The convex hulls at intermediate levels of the tree are constructed from the convex hulls of the two linearly separated child nodes. This operation is linear in the number of hull points, making this part of the pre-processing phase very fast. As Dadoun and Kirkpatrick [13] point out, the hierarchy thus constructed allows us to exploit convexity even in highly nonconvex environments. It can greatly accelerate beam intersection testing by rejecting objects in clusters rather than individually, which is exactly analogous to the bounding volume hierarchy techniques of Section 4.

The hierarchical scene representation is a binary tree of convex hulls separated by planar partitions. Given a beam origin, the partitions provide an efficient means of assigning a priority to the groups of enclosed objects in exactly the same way that polygons are prioritized in the BSP hidden surface algorithm [16]. This moves much of the sorting operation into the initial construction of this data structure. The recursive traversal algorithm shown in *Figure 12* is based upon the same principle and requires little modification to be applicable in this context. The result is similar to that achieved by the algorithm of Kay and Kajiya shown in *Figure 9*. Hierarchically nested convex hulls are examined in the order in which they are encountered by the beam.

To further accelerate the beam–hull intersection testing, Dadoun and Kirkpatrick [13] augment the beam and each convex hull of the hierarchy with an *outer sequence* [12]. This is a nested sequence of successively large and simpler convex polyhedra which are formed by removing bounding half-spaces. Given two polyhedra with n and m hull points, respectively, if both are augmented by an outer sequence the intersection test can be done in $O(\log(m) + \log(n))$ time [12].

A pencil is another type of generalized ray. It is comprised of rays which are in the vicinity of a special ray called the *axial ray*. Each of these nearby *paraxial* rays can be represented as a 4-D vector encoding its deviation from the axial ray. Shinya *et al.* [54] used techniques from paraxial approximation theory to determine how these pencils interact with surfaces encountered in an environment, viewing it as an optical system. By restricting attention to small deviations from the axial ray, the pencil transformations could be assumed to be linear, and therefore representable as 4×4 *system matrices*. Propagation of rays grouped as pencils could then be carried out by combining the system matrices corresponding to the individual surfaces. The approximation is only valid for sufficiently smooth surfaces, however, so it can only be applied to pencils which do not encounter edges or surface discontinuities. Shinya *et al.* traced individual rays in the areas which posed these problems.

10 OPTIMIZATIONS FOR CSG

Using the method of CSG (Constructive Solid Geometry), solid objects are represented by combining primitive solids with the boolean set operators *intersection* (&), *union* (+), and *difference* (−). One way to generate shaded images from a CSG model is to generate a boundary surface representation from the model and then render those surfaces using some hidden-surface algorithm. In contrast, ray tracing can generate images of CSG models by intersecting rays directly with the CSG tree. A straightforward method for intersecting rays with CSG trees is to 'classify' each ray against the CSG tree, determining the intervals along the ray which intersect the solid. Roth [50] described this process as a recursive walk down the tree structure, intersecting the ray with each primitive in the tree, and combining the resultant intervals on the way back up the tree.

This algorithm can be accelerated by the use of object bounding volumes. Roth [50] discussed the application of 2-D box and 3-D sphere bounding volumes in this manner and reported a factor-of-two speed-up. The 2-D box aids in the first-generation rays only, but the sphere may be used for other rays as well. In addition, the ray distance interval optimization described in Section 4 can be used to eliminate some CSG subtrees from consideration.

Gervautz [19] used both of these techniques and also applied 3-D bounding boxes within the tree to reduce the number of ray/primitive intersection checks.

It is possible to reduce the number of intersections by using known characteristics of the CSG operators. In the case of union (+), the tree can be rearranged without affecting the root object. Gervautz [19] pointed out that this can be advantageous in terms of reducing the size of aggregate bounding volumes. One can also take advantage of potential subtree elimination with the ' – ' and '&' operators [50]. For instance, in the combination 'A–B,' if the ray does not intersect 'A,' there is no value in checking the 'B' subtree for intersection with the ray, since it cannot affect the outcome.

Since the process of ray–CSG tree intersection is recursive, it is advantageous to reduce the overhead typically associated with recursion, such as procedure calls and dynamic memory allocation [50]. Unfortunately, the organization of the task is such that the classification and combination of intervals must be performed independently for each ray, regardless of any coherence which exists. Also, the entire set of 1-D ray intervals must be computed since it is not known *a priori* which will be the closest. The classification, which is essentially a depth sorting operation, must be performed on all of the intervals. Atherton [4] proposed using a hybrid scan-line/ray tracing algorithm to solve this problem. The primitives in the CSG tree are decomposed into polygonal approximations, and a Y–X–Z scan-line algorithm is applied. Spans are maintained which represent simplifications of the original CSG tree. At each pixel, the CSG problem is only solved for the first intersection.

Bronsvoort *et al.* [5] described an alternative way to utilize the coherence properties of scan-line algorithms. The ray-bounding volume check described by Roth [50] is replaced by a point/scan-line interval test. At each scan line, only part of the CSG tree may contribute to the image. It is possible, therefore, to maintain an *active subtree* of the CSG tree which is analogous to the active polygon list in a typical scan-line algorithm. This greatly reduces the complexity of the ray–CSG tree intersection which must be performed. Maintaining a list of intervals instead of a hierarchy of bounding boxes is more efficient because the intervals represent a tighter bound. The performance improvement is lessened due to the extra cost of computing and sorting the intervals. A greater gain is realized by the simplification of the CSG tree which can be performed.

Gervautz [19] also created 'temporary' active subtrees to accelerate the ray tracing process. The subtrees for first-generation rays are created by projecting the bounding volumes of primitive objects onto the view plane and maintaining a quadtree structure. Each pixel in the quadtree can be associated with those objects which penetrate it. In order to accelerate the tracing of shadowing rays, another quadtree can be generated for a projection plane from a point light source. For other rays (reflection and transparent), an octree was used. The CSG object tree must be quite complicated before the savings in ray tracing time are negated by the cost of constructing the subtrees, particularly in the octree case.

Youssef [71] described a variation of CSG in which the objects are restricted to interval representations in some coordinate space. Examples of such objects are boxes in Cartesian coordinates, spheres in spherical coordinates, and cylinders in cylindrical coordinates. Aggregate objects are constructed by combining the coordinate spaces using the union (+) or subtraction (−) operators. Intersection (&) is not provided. The process of ray tracing is carried out by tracing rays through the *coordinate volumes* in which the objects are represented. This approach is most effective when representing many regularly spaced similar objects, such as bricks in a wall.

Wyvill *et al.* [68] also considered the use of an octree to subdivide space. Within each voxel of the octree, the space can be classified with respect to the solid represented by the CSG tree (or DAG). The possible classifications are: (1) full (completely contained within some solid); (2) empty (completely outside all solids); (3) contains boundaries between empty space and one primitive object; (4) contains boundaries between full space and one subtracted primitive object; (5) volume is below some minimum size threshold.

Cases (1)–(4) are straightforward to ray trace directly, but case (5) is more complex. In their early paper [66], Wyvill and Kunii labeled these voxels as 'nasty cells' and either ignored them or colored them black. In [68] each of the voxels in case (5) is represented as a pruned CSG subtree which contains only those primitives which are present in that voxel. The subtrees are constructed as part of the process of generating the octree. The full CSG tree is traversed, subdividing each primitive into its component octree voxels. The different octree structures are then combined according to the CSG operators linking them together. The CSG tree simplification is implicit in this process in which only relevant primitives ever appear in a given voxel. In the process of ray tracing, the octree structure is traversed and only those subtrees which are encountered are intersected with the rays. This simplification of a CSG tree into smaller subtrees is a theme which recurs in almost all of the approaches for ray tracing acceleration in the context of CSG.

Fujimoto *et al.* [18] described a similar approach, but used the SEADS approach described in Section 5.2. The task of filling this data structure was performed by a pre-processor termed B-COM, for 'boolean compiler.' The B-COM classified voxels as being either *homo* or *hetero*. The homo case corresponds to cases (1) and (2) from [68], and the hetero case corresponds to cases (3), (4) and (5). In the process of tracing rays, all homo voxels can be ignored and rays need only be intersected with the contents of hetero voxels which the ray pierces. Fujimoto's results indicate that SEADS outperforms the octree method if the pre-processing time is ignored.

Arnaldi *et al.* [2] also described a voxel structure within which simplified CSG subtrees can be used to accelerate ray intersection calculations. The structure is hierarchical but not regular. The image plane is divided into parallelepiped cells which closely surround projections of the primitive objects. These cells are constructed around the bounding boxes of the primitives and intermediate nodes. Some minimization of the bounding boxes at the intermediate intersection (&) nodes is possible by computing the intersections of the bounding boxes. A Binary Space Partitioning (BSP) algorithm is used to perform the subdivision and classification of bounding boxes with respect to the voxels.

After the 2-D partitioning has been performed, each resulting frustrum is subdivided in depth to generate a stack of frustra. This presents some difficulty in determining which voxel is a neighbor when a ray passes from one column of voxels to another. To accelerate the voxel walking, a set of pointers is maintained to express the connectivity. The adjacency is expensive to compute, but the cost can be reduced by calculating it as subdivision proceeds and updating it continuously. Tracing first-generation rays is efficient because each ray traverses a single column. Other rays are much more costly due to the added expense of calculating the neighbor across columns.

The main optimization which underlies all of the approaches for accelerating ray tracing of CSG is to subdivide space and pre-process or compile the CSG structure into the spatial hierarchy. Subtrees of the main CSG tree can be generated for individual spatial hierarchy nodes by intersecting the volume of the node (or voxel) with the CSG tree. This process allows the ray tracer to take advantage of the coherence present due to the locality of primitives.

11 PARALLELIZATION AND VECTORIZATION

Acceleration of ray tracing can also be achieved by performing some of the operations concurrently. Several approaches to this have been attempted, including: (1) vectorization; (2) execution on a collection of general-purpose computers; (3) execution on a general-purpose multicomputer; and (4) custom special-purpose hardware.

In addition, a number of parallel algorithms have been developed which are broadly applicable. We will discuss the different classes independently, progressing in roughly chronological order. Approach (2), execution on a collection of general-purpose computers, has not really been directly addressed in any research, although it clearly has a place in this taxonomy. Although we will not explicitly discuss this case, it is interesting to note that many of the special-purpose architectures are described in terms of simulated performance on one or more general-purpose computers.

11.1 Vectorization

Max [43] organized a restricted procedural ray tracer for the vectorizing compiler on a Cray-1 supercomputer. The procedural model rendered ocean waves and islands. The waves are represented as a height field constructed from superimposed traveling sine waves. The islands are also represented as height fields composed of elliptical paraboloids with superimposed cosine terms to give rolling hills. First-generation rays are traced against the water and islands, as are up to two reflections from the water. No shadow rays are traced, although island surfaces which face away from the light source (sun) are considered to be in shadow. This relatively uniform organization allows the ray tracing to be vectorized more efficiently.

The ocean height field points which are relevant for a given scan line are bounded by an ellipse. Using this bound, only a subset of the possible points must be generated. The set of points is passed through a depth-buffer to determine the visible points. The first-generation rays are processed as a vector, and the resulting shading calculations are also vectorized. Similarly, those rays which are reflected are gathered into smaller vectors to be processed as a unit.

Plunket and Bailey [49] described a more general implementation of ray tracing on a CDC Cyber 205. The task is organized to trace a list of rays sequentially against all of the surfaces in the scene. The list accumulates until it is large enough to be traced, and each ray is considered concurrently and totally independently. In other words, there is no advantage taken of coherence between rays. A simple implementation of vectorized ray tracing was described as follows.

While there are still unfinished pixels:

(1) Add first-generation, reflection, and shadowing rays to the queue until it is full.
(2) Intersect the entire queue of rays with each surface in the scene using vector code.
(3) Determine the visible surface for each ray using CSG evaluation techniques.
(4) Spawn additional rays for modeling special effects and add these to the queue.
(5) Determine the intensity of pixels which have complete visible surface calculations.

This algorithm is necessarily more complex than the scalar version since more than one pixel is being processed at the same time. An additional problem is that given that CSG operations are going to be performed based on the intersection distances, the results of processing the queue must include enough information to resolve the CSG tree. The storage space required for results is proportional to the product of the number of rays and the number of objects. This conflicts with the desire to make the vector queue as long as possible to most effectively use the vector capabilities. A compromise is to process rays in groups of 500.

The process of traversing the CSG tree is also vectorized because the time spent in this operation becomes significant once the intersection calculations are vectorized. Vectorizing the tree traversal requires that the tree be traversed in the same order for all rays and therefore precludes the subtree simplifications of many other CSG algorithms. Though more arithmetic operations are required in the vectorized organization, there is a net gain in performance due to the absolute speed of the vector processing.

11.2 Special-purpose Hardware

An example of special purpose ray tracing hardware is the LINKS-1 multicomputer [46]. This is a rare specimen among special-purpose rendering architectures because it has actually been built and is in operation generating ray traced images. LINKS-1 is similar to the vectorized approaches in that each ray is traced concurrently and independently. The LINKS-1 architecture consists of 64 *node computers* interconnected with a single controlling root computer. The root computer can dynamically reconfigure the organization of the node computers, using them in parallel, as a pipeline, or in any combination. Communication between node computers is achieved through an *intercomputer memory swapping unit*, which is a device for transferring large amounts of data between node computers.

Each node computer consists of a Zilog Z8001 control processor (CU), an Intel 8086/8087 arithmetic processor (APU), 1 Mbyte of local memory, and is attached to two intercomputer memory swapping units (IMSU). The APU operates as a slave of the CU. Each node computer N(i) is connected to its nearest neighbor N(i + 1) by an IMSU. It is also connected to the root computer via another IMSU. These connections allow rapid swapping of data between processors.

The process of ray tracing on the LINKS-1 is described as a pipelined sequence of object sorting, ray tracing, and shading. The node computers can be configured as a set of parallel pipelines to render a sequence of images. It is assumed that each pipeline retains the entire world database, and rays are distributed among different pipelines. Timings for execution of such a configuration provided parallel utilization of up to 65%, largely because of the ray tracing component of the pipe which is kept busy. However, the first and third stages of the pipe (object sorting and shading, respectively) are often kept waiting.

While the LINKS architecture duplicates the entire database and distributes rays, Kobayashi *et al.* [41] proposed a parallel machine in which the database is distributed over a set of intersection processors (IPs). Each IP receives only a portion of the world which corresponds to spatial subdivision and rays are passed from one processor to another as they are propagated through space. A host computer generates the initial viewing rays and distributes them to the appropriate IP based on the ray direction. Each IP checks its rays for intersection with its objects and passes on the rays which do not intersect anything. Each IP is also responsible for calculation of the next IP. Rays which do intersect an object are passed to a network of shading processors (SP) which resolve the ray tree and generate final pixel colors.

The space bounding the environment is subdivided using an octree. To ease the problem of stepping between voxels of varying size, a quadtree is maintained on each voxel face to keep track of the neighbors. The octree is first generated based on the distribution of objects, and then the *face-neighbor quadtree* is constructed. This method was termed an *adaptive division graph*. The resulting space was mapped onto a 6-D hypercube computer, allowing nearest-neighbor communication for face-adjacent voxels of identical size. Due to the sixfold connectivity of a hypercube, neighbors at different levels of the octree are also close in terms of the number of message hops. Timing results were presented for a 512 × 512 image, ignoring the time for constructing the subdivided space. The ray tracing time for environments containing between 1 and 4096 objects was found to be almost constant. The performance for the adaptive division graph appeared to be better than both regular grid subdivision and a normal octree without the quadtree face-neighbor structure.

Dippé and Swenson [15] also described an adaptive subdivision algorithm and a parallel architecture for ray tracing. The world was initially subdivided into a regular grid in three dimensions, which was mapped on to a hypothetical 3-D array of processors. Viewing rays are generated by the processor responsible for the region containing the eye, and are propagated to other processors based on their paths. The load at each processor, defined as the product of the number of rays to and the number of objects, was used as a metric for redistribution of objects and rays. The redistribution was achieved by relaxing the requirements of regularity and reshaping the voxels. Several different geometries for the voxels were discussed, including orthogonal parallelepipeds, general cubes, and tetrahedra.

The orthogonal case was dismissed because it does not allow for local redistribution. A local redistribution request forces other regions to shift regardless of the utility of shifting them. Tetrahedra were also dismissed since they can too easily become inappropriately shaped for bounding objects. The case of general cubes was used for most of the discussion. General cubes or loosely termed hexahedra are regions with six (possibly non-planar) faces, six neighbors, and eight vertices. Loads are transferred between regions when a processor determines that its load is greater than its neighbors by more than a fixed threshold. The shift was accomplished by moving one vertex and then reshuffling the objects and rays based on the new region shape.

A proposed implementation of this mechanism would have a 3-D array of autonomous computers, which communicate by passing messages. Computers on the edge would be connected to frame buffers, disks and other peripherals. A preliminary analysis of the performance of the algorithm suggests an upper bound of $O(S^{2/3})$, where S is the number of processors. A problem which arises when load is transferred by moving a vertex is that eight regions are affected. In order to do well with this redistribution, we must efficiently determine which vertex to move by how much and in what direction. This is a nontrivial problem. Also, in order to redistribute objects, we must intersect them with the boundaries of these general cubes, and intersect rays with them.

Nemoto and Omachi [45] attempted to address some of these problems. They simulated a similar 3-D processor grid using a simpler redistribution algorithm. The basis of the approach was to subdivide space using a regular grid structure (Fujimoto *et al.* [18]) and distribute the voxels to different processors. Each processor generates a portion of the viewing rays and passes them to the appropriate processor for intersection checking. Rays are propagated efficiently between processors using a variant of Fujimoto's 3DDDA. The redistribution was performed only along one axis. It was determined which axis had the most variation in number of objects and the boundaries of the voxels were allowed to 'slide' along this 'driving axis.' Redistribution occurred when the load, defined as running time/idle time was determined to be above a given threshold compared to the neighbors along the driving axis. The object intersection with the new boundaries was a simple plane intersection check, and ray propagation was only slightly complicated from the normal 3DDDA case. After the 3DDDA step, an adjustment might have to be made along the driving axis to find the correct voxel.

This method achieved far greater efficiency in tracing rays and speed of load balancing at the cost of less effective load balancing. A software simulation of this algorithm operating on a 1, 8, 64, and 512 processor version of this architecture showed reasonable performance improvements when redistribution was used, as compared to a normal 3-D grid spatial subdivision. The measurements indicated very near linear performance increases for multiple processors, when the scene complexity was high.

Cleary *et al.* [7] independently analyzed the performance of ray tracing with 2-D and 3-D space subdivisions on multiprocessor systems. No attempt was made at load balancing, but a detailed performance analysis was offered. The analysis was performed for an empty scene, assuming that the intersection times for real scenes would scale down with larger numbers of processors. The upper bound for the speedup of a 3-D network was given as $N^{2/3}$, and for a 2-D network varied from N to $N^{1/2}$ as the number of processors increased. The conclusion was that given a small number of processors, a 2-D spatial subdivision may be more efficient. Simulation was performed for a number of processors varying from 1 to 10 000, and a number of objects varying from 1 to 8.

Ullner [60] examined the mathematics involved in the actual task of intersecting rays with objects, specifically convex quadrilaterals. A ray tracing peripheral was described which used special purpose hardware to intersect the environment of polygons with each ray. This task was decomposed into three stages: fetching each polygon, computing the distance to the point of intersection (if one exists), and comparing these distances to find the nearest one. These three stages can be pipelined and each stage can be further pipelined and parallelized. Exhaustive ray tracing was performed using one or more of these peripherals. The performance of one of these theoretical devices was quite impressive (for 1983), being able to compute a new ray–polygon intersection every 1/3 microsecond once the pipe was full. This is comparable to the speed of a CRAY-1 supercomputer. It would thus be able to exhaustively ray trace an anti-aliased 512×512 image containing 1000 polygons in approximately 10 minutes.

Ullner also observed that a 3-D regular grid subdivision could be applied, producing commensurate performance improvements. This additional intelligence of walking voxels and retrieving only a subset of polygons is beyond the scope of the original hardware and requires an additional processor. This processor uses the voxel data structure to determine which polygons may potentially intersect the ray, and passes them to the pipeline containing the ray. Using a test scene of 1000 polygons, Ullner's simulation achieved optimal results with a grid of approximately $11 \times 11 \times 11$ voxels. Other test scenes produced similar results.

Another parallel hardware approach suggested by Ullner involved massive use of VLSI circuits. A relatively slow (5 ms/intersection) ray–polygon intersection processor could be implemented on a chip. A large scale machine could be constructed by stringing together a large number of such chips in a pipeline.

A more practical solution, also described by Ullner, is to use a 2-D grid of special purpose intersection processors to implement the 3-D regular grid subdivision described above. In order to balance the load more evenly between these processors, it was suggested that the voxels can be shifted so that a given processor is responsible for a stairstep pattern of voxels, instead of a slab.

11.3 General-purpose Multicomputers

Goldsmith and Salmon [22] described an actual implementation of a ray tracer on a hypercube. A hypercube is a multicomputer with 2^N processors connected with the topology of an N-dimensional hypercube. Messages are passed between processors via these connections and it may require several hops to get from one processor to another. An important property of the hypercube topology is that no more than N hops are ever required to pass a message from one processor to another. Thus, in a hypercube of dimension 5, there are 32 processors, each of which is directly connected to 5 other processors, and the most widely separated processors are a 'distance' of 5 hops apart. Frequently the message passing speed is much slower than the processing speed, so it is important to minimize interprocess communication. In Goldsmith's approach, first-generation rays are treated as being completely independent and are distributed among the processors of the hypercube with no interaction between rays. Therefore, rays traced on one processor do not affect the outcome of rays traced on any other processor.

Two basic methods are described in [22]. One involves replicating the entire database and distributing rays, while the other involves partitioning the database among the processors and distributing the rays. In the first method, assuming that the entire database is replicated in each processor, one must decide on the optimal distribution of rays between processors. A scattered decomposition is preferred because it balances the load of sparse and difficult pixels among the processors. A disadvantage is that in anti-aliasing, the nearest neighbors are important and are not readily available in the scattered decomposition. Therefore, it is reasonable to switch from a scattered to a regular decomposition after the intersections have been computed.

In the second method, a bounding volume hierarchy is distributed among multiple processors. Each processor maintains the top few levels of the hierarchy, but the lower (larger) structures are distributed among the processors. Each processor maintains the following data structures: (1) its subset of the pixel array; (2) the top of the bounding volume hierarchy (known to all processors); (3) one or more subtrees of the hierarchy (private to this processor); (4) the database for the background (known to all processors).

This method is reasonable assuming that the communication time does not overwhelm the actual computation. Because of the hypercube connection scheme, the time to communicate between two processors is proportional to the number of bits which differ between the processor numbers. Therefore, the average time can be reduced by duplicating subtrees into processors whose ids are 1's complements of each other. If each subtree connection chooses the nearest neighbor of the two, the average communication time is cut in half.

Another potential optimization described by Goldsmith and Salmon [22] is that the load between processors can be adaptively balanced by processors which complete early and request additional work. However, this increases communication requirements and could actually degrade performance. This approach was not implemented, so there are no results to indicate its effectiveness.

12 COMBINING OPTIMIZATIONS

The spatial subdivision methods described in Section 5 simply change one large problem into many small problems which are typically handled by exhaustive ray tracing. This needn't be the case. Bounding volumes may still be appropriate within voxels, as well as virtually any other optimization technique. Because there may be a large number of these reduced problems, and more than one may be encountered per ray, the start-up overhead and space requirements of the techniques applied to them must be minimal. A number of recent contributions have addressed the idea of combining several acceleration techniques to gain some of the benefits of each. By constructing a hierarchy comprised of several different techniques, the performance can be superior to that of any individual technique.

Snyder and Barr [55] compared the performance of uniform 3-D spatial subdivision, octree-based nonuniform subdivision, bounding volume hierarchies, and lists of primitive objects. They observed that for large numbers of homogeneously distributed objects of similar scale, a regular grid outperforms octree methods due to the efficiency of voxel walking. In the event that the primitive objects are not all of the same scale or are unevenly distributed, the regular divisions become costly. A compromise involves defining an object abstraction which allows primitive objects, regular 3-D grids, and lists to be handled similarly. This is achieved by defining a C-language structure for these objects in which a representation of any one of these constructs may be stored. Each of the elements of the environment hierarchy may be one of these objects. In this way the environment can be constructed from a hierarchy of regular 3-D grids, lists and primitive objects. Through instancing, this allowed Snyder and Barr to render environments containing billions of objects.

Scherson and Caspary [52] described a similar mechanism. By analyzing the complexity of ray tracing several environments, they were able to identify some general situations in which octrees are likely to outperform bounding volume hierarchies, and other cases in which the reverse is true. They concluded that octrees performed well in cases where the the first intersection was likely to be found near the perimeter of the environment, before many voxels had been processed. The costs associated with fragmentation (objects appearing in more than one voxel) and voxel walking diminished effectiveness if most of the intersections were found deep within the environment. In the latter case, bounding volume hierarchies were found to be superior. Scherson and Caspary found that a compromise between these methods was effective. They created a hybrid structure in which the top levels of the hierarchy are formed by octree spatial subdivision while the lower levels consist of bounding volume hierarchies. The method described by Kay and Kajiya [37] was used in the latter case.

Glassner [21] developed a different approach which combines advantages of both bounding volume hierarchies and nonuniform spatial subdivision. Glassner observed that bounding volumes can offer tight bounds but usually produce hierarchies in which the volumes overlap, thereby decreasing overall efficiency. On the other hand, nonuniform spatial subdivision produces hierarchies in which the volumes are disjoint, though they do not provide tight object bounds. By using nonuniform space subdivision to guide the construction of a bounding volume hierarchy, volumes can be selected which are both tight fitting and disjoint. This is done by top-down construction of an octree followed by a bottom-up construction of bounding volumes based on the plane-set approach of Kay and Kajiya [37]. Each bounding volume is chosen to tightly enclose only the portions of the objects which are within each voxel. The bounding volumes are therefore guaranteed to be disjoint because they are contained within disjoint voxels. This is quite similar to the approach described by Dadoun and Kirkpatrick [13] for constructing a hierarchical scene representation. Glassner also generalized this technique to four dimensions representing space-time in order to accelerate the creation of animated sequences complete with motion blur. Instead of ray tracing objects which are moving in 3-D space, the technique processes static objects in 4-D space-time.

A very general mechanism for combining optimizations was described by Kirk and Arvo [39]. In their approach, acceleration techniques are encapsulated in such a way that they present essentially the same interface as procedurally defined primitive objects. That is, an acceleration technique becomes an *aggregate object* which is responsible for computing ray intersections with a collection of subordinate objects, or children. The children may include other aggregate objects as well as primitive objects. By adhering to a uniform procedural interface among all primitive and aggregate objects, it is possible to create *meta hierarchies* consisting of any number of diverse algorithms including octrees, uniform grids, bounding volume hierarchies, and directional techniques. Though the nesting of different techniques through this mechanism carries additional costs which might be avoided by special-purpose implementations, Kirk and Arvo found that its flexibility allowed easy experimentation and proved to be effective in constructing and rendering environments containing millions of objects.

13 FUTURE DIRECTIONS

As mentioned at the outset, there is currently a lack of meaningful quantitative comparisons among acceleration techniques. A fertile area for future research is to provide quantitative techniques for performing such analysis. This will require more sophisticated statistical tools which can accurately predict average case behavior in very complex environments. Such tools may also provide an important ingredient for more intelligent algorithms which are 'self-tuning' and can adjust to the local complexity of the environment. This may prove to be the next logical step beyond automatic construction of bounding volume hierarchies. Goldsmith and Salmon's [23] work has provided an excellent start in terms of characterizing how well a given hierarchy will perform. They also achieved good results without resorting to an exhaustive search of all of the possibilities. Perhaps methods such as simulated annealing [40] can be applied in more complex settings to achieve similar results.

New areas of research such as directional techniques have only begun to be investigated. Thus far these techniques have improved performance only at the cost of greatly increased storage requirements. More investigation is needed to determine if this is an inherent shortcoming or merely a drawback of the particular implementations which have been explored. Storage reduction may be achievable through hybrid algorithms which combine features of directional techniques with bounding volume hierarchies or spatial subdivision techniques.

The rendering equation [34] poses new problems for acceleration. Since the role of ray tracing in the solution of the rendering equation involves random walks, many of the acceleration techniques will fail to take advantage of environment coherence. Efficient solution will require new variance reduction techniques which can characterize the energy balance in the environment and increase the statistical efficiency of the random paths which are traced. Conversely, the use of techniques which rely heavily on ray coherence can be extended to have a wider application to acceleration. Generalized rays such as cones [1, 38], beams [13, 30] and pencils [54] can be used to characterize sets of rays before application of other techniques.

Another opportunity exists in the area of parallelization and concurrent execution of ray tracing. The only approaches which have made efficient use of multiple processors to date have distributed first-generation rays to different processors, each supplied with a complete copy of the environment. This is unreasonable for extremely complex environments and wasteful of coherence information which could further accelerate the process. Algorithms which utilize multiple processors yet minimize both storage and communication requirements need to be developed.

ACKNOWLEDGMENTS

We wish to thank Christian Bremser, John Francis, Semyon Nisenzon, Ken Severson, Cary Scofield, and in particular Douglas Voorhies for their assistance in preparing this paper.

REFERENCES

1. Amanatides, J., Ray tracing with cones. *Comput. Graph.* **18**(3), 129–135, July 1984.
2. Arnalldi, B., Priol, T. and Bouatouch, K., A new space subdivision method for ray tracing CSG modelled scenes. *The Visual Computer*, Springer-Verlag, Vol. 3, pp. 98–108, 1987.
3. Arvo, J. and Kirk, D., Fast ray tracing by ray classification. *Comput. Graph.* **21**(4), 55–64, July 1987.
4. Atherton, P., A scan-line hidden surface removal procedure for constructive solid geometry. *Comput. Graph.* **17**(3), 73–82, July 1983.
5. Bronsvoort, W.F., van Wijk, J.J. and Jansen, F.W., Two methods for improving the efficiency of ray casting in solid modeling. *Comput. Aided Design* **16**, 51–55, 1984.
6. Clark, J.H., Hierarchical geometric models for visible surface algorithms. *Commun. ACM* **19**(10), 547–554, October 1976.
7. Clearly, J.G., Wyvill, B.M., Birtwistle, G.M. and Vatti, R., Multiprocessor ray tracing. *Comput. Graph. For.* **5**, 3–12, 1985.
8. Cohen, M.F., and Greenberg, D.P., The hemi-cube: a radiosity solution for complex environments. *Comput. Graph.* **19**(3), 31–41, July 1985.
9. Cook, R.L., Porter, T. and Carpenter, L., Distributed ray tracing. *Comput. Graph.* **18**(3), 137–145, July 1984.
10. Cook, R.L., Stochastic sampling in computer graphics. *ACM Trans. Graph* **5**(1), January 1986.
11. Coquillart, S., An improvement of the ray-tracing algorithm. *Proceedings Eurographics '85* (ed. C.E. Vandoni), Elsevier (North-Holland), pp. 77–88, 1985.
12. Dobkin, D.P. and Kirkpatrick, D.G., Fast detection of polyhedral intersection. *Theor. Comput. Sci.* **17**, 241–253, 1983.
13. Dadoun, N. and Kirkpatrick, D.G., The geometry of beam tracing. *Proc. of the Symposium on Computational Geometry*, pp. 55–61, June 1985.
14. Dippe, M. and Swensen, J., An adaptive subdivision algorithm and parallel

architecture for realistic image synthesis. *Comput. Graph.* **18**(3), 149–158, July 1984.

15. Dippe, M. and Wold, E.H., Antialiasing through stochastic sampling. *Comput. Graph.* **19**(3), 69–78, July 1985.
16. Fuchs, H., On visible surface generation by a priori tree structures. *Comput. Graph.* **14**(3), 124–133, July 1980.
17. Fujimoto, A., Tanaka, T. and Iwata, K., ARTS: Accelerated Ray-Tracing System. *IEEE Comput. Graph. Appl.* **6**(4), 16–26, April 1986.
18. Fujimoto, A., Perrott, C.G. and Iwata, K., Environment for fast elaboration of constructive solid geometry. *Adv. Comput. Graph.* (Proc. of Computer Graphics Tokyo '86), 20–32, April 1986.
19. Gervautz, M., Three improvements of the ray tracing algorithm for CSG trees. *Comput. Graph.* **10**(4), 333–339, 1986.
20. Glassner, A.S., Space subdivision for fast ray tracing. *IEEE Comput. Graph. Appl.* **4**(10), 15–22, October 1984.
21. Glassner, A.S., Spacetime ray tracing for animation. *IEEE Comput. Graph. Appl.* **8**(3), 60–70, March 1988.
22. Goldsmith, J. and Salmon, J., A ray tracing system for the hypercube. Caltech Concurrent Computing Project Memorandum HM154, California Institute of Technology, 1985.
23. Goldsmith, J. and Salmon, J., Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.* **7**(5), 14–20, May 1987.
24. Goldstein, R.A. and Nagel, R., 3-D visual simulation. *Simulation* **16**(1), 25–31, January 1971.
25. Haines, E.A. and Greenberg, D.P., The light buffer: a shadow testing accelerator. *IEEE Comput. Graph. Appl.* **6**(9), 6–16, September 1986.
26. Haines, E., A proposal for standard graphics environments. *IEEE Comput. Graph. Appl.* **7**(11) 3–5, November 1987.
27. Hall, R.A. and Greenberg, D.P., A testbed for realistic image synthesis. *IEEE Comput. Graph. Appl.* **3**(10), 10–20, November 1983.
28. Hanrahan, P., Ray tracing algebraic surfaces. *Comput. Graph.* **17**(3), 83–89, July 1983.
29. Hanrahan, P., Using caching and breadth-first search to speed up ray-tracing, *Proc. of Graphics Interface '86*, Vancouver, B.C., 56–61, May 1986.
30. Heckbert, P.S. and Hanrahan, P., Beam tracing polygonal objects. *Comput. Graph.* **18**(3), 119–127, July 1984.
31. Jansen, F.W., Data structures for ray tracing. In *Data Structures for Raster Graphics, Proceedings Workshop* (eds L.R.A. Kessener, F.J. Peters, M.L.P. Lierop) pp. 57–73, Eurographics Seminars, Springer Verlag, 1986.
32. Joy, K.I. and Bhetanabhotla, M.N., Ray tracing parametric surface patches utilizing numerical techniques and ray coherence. *Comput. Graph.* **20**(4), 279–285, August 1986.
33. Kajiya, J.T., New techniques for ray tracing procedurally defined objects. *Comput. Graph.* **17**(3), 91–102, July 1983.
34. Kajiya, J.T., The rendering equation. *Comput. Graph.* **20**(4), 143–150, August 1986.
35. Kaplan, M.R., Space tracing a constant time ray tracer. State of the Art in Image Synthesis (Siggraph '85 Course Notes), Vol. 11, July 1985.
36. Kaplan, M.R., The use of spatial coherence in ray tracing. In *Techniques for Computer Graphics* (eds David F. Rogers and Rae A. Earnshaw), Springer-Verlag, New York, 1987.

37. Kay, T.L., and Kajiya, J., Ray tracing complex scenes. *Comput. Graph.* **20**(4), 269–278, August 1986.
38. Kirk, D.B., The simulation of natural features using cone tracing. *The Visual Computer*, Springer-Verlag, Vol. 3, No. 2, pp. 63–71, 1987.
39. Kirk, D. and Arvo, J., The ray tracing kernel. *Proc. of Ausgraph '88*, Melbourne, Australia, pp. 75–82, July 1988.
40. Kirkpatrick, S., Gelatt, Jr., C.D., and Vecchi, M.P., Optimization by simulated annealling. *Science* **220**, 671–680, 13 May 1983.
41. Kobayashi, H., Nakamura, T. and Shigei, Y., Parallel processing of an object space for image synthesis using ray tracing. *The Visual Computer*, Springer-Verlag, Vol. 3, No. 1, pp. 13–22, 1987.
42. Lee, M., Redner, R.A. and Uselton, S.P., Statistically optimized sampling for distributed ray tracing. *Comput. Graph.* **19**(3), 61–67, July 1985.
43. Max, N.L., Vectorized procedural models for natural terrain: waves and islands in the sunset. *Comput. Graph.* **15**(3), 317–324, August 1981.
44. Muller, H., Imge generation by space sweep. *Comput. Graph. For.* **5**, 189–196, 1986.
45. Nemoto, K. and Omachi, T., An adaptive subdivision by sliding boundary surfaces. *Proc. of Graphics Interface '86*, Vancouver, B.C., pp. 43–48, May 1986.
46. Nishimura, H., Ohno, H., Kawata, T., Shirakawa, I. and Omura, K., LINKS-1: a parallel pipelined multimicrocomputer system for image creation. *Proc. of the 10th Symposium on Computer Architecture*, SIGARCH, pp. 387–394, 1983.
47. Ohta, M. and Maekawa, M., Ray coherence theorem and constant time ray tracing algorithm. *Computer Graphics 1987* (Proc. of CG International '87) (ed. T.L. Kunni), pp. 303–314.
48. Purgathofer, W., A statistical method for adaptive stochastic sampling, *Proc. Eurographics '86* (ed. A.A.G. Requicha), Elsevier (North-Holland), pp. 145–152. 1986.
49. Plunkett, D.J. and Bailey, M.J., The vectorization of a ray-tracing algorithm for improved execution speed. *IEEE Comput Graph. Appl.* **5**(8), 52–60, August 1985.
50. Roth, S.D., Ray casting for modeling solids. *Comput. Graph. Image Process.* **18** 109–144, 1982.
51. Rubin, S. and Whitted, T., A three-dimensional representation for fast rendering of complex scenes. *Comput. Graph.* **14**(3), 110–116, July 1980.
52. Scherson, I.D. and Caspary, E., Data structures and the time complexity of ray tracing. *The Visual Computer*, Springer-Verlag, Vol. 3, pp. 201–213, 1987.
53. Sederberg, T.W. and Anderson, D.C., Ray tracing of Steiner patches. *Comput. Graph.* **18**(3), 159–164, July 1984.
54. Shinya, M., Takahashi, T. and Naito, S., Principles and applications of pencil tracing. *Comput. Graph.* **21**(4), 45–54, July 1987.
55. Synder, J.M. and Barr, A.H., Ray tracing complex models containing surface tessellations. *Comput. Graph.* **21**(4), 119–126, July 1987.
56. Speer, L.R., DeRose, T.D.,and Barsky, B.A., A theoretical and empirical analysis of coherent ray tracing. *Computer-Generated Images* (Proc. of Graphics Interface '85), 27–31 May 1986, pp. 11–25.
57. Sutherland, I.E., Sproull, R.F. and Schumacker, R.A., A characterization of ten hidden-surface algorithms. *Comput. Surv.* **6**(1), 1–55, March 1974.
58. Sweeney, M.A.J. and Bartels, R.H., Ray tracing free-form B-spline surfaces. *IEEE Comput. Graph. Appl.* **6**(2), 41–49, February 1986.

59. Toth, D.L., On ray tracing parametric surfaces. *Comput. Graph.* **19**(3), 171–179, July 1985.
60. Ullner, M.K., Parallel machines for computer graphics. Ph.D. Dissertation, California Institute of Technology, Pasadena, California, 1983, 5112:TR:83.
61. van de Hulst, H.C., *Light Scattering by Small Particles*, Dover Publications, New York, 1981.
62. van Wijk, J.J., Ray tracing objects defined by sweeping planar cubic splines. *ACM Trans. Graph.* **3**, 223–237, (3), July 1984.
63. Weghorst, H., Hooper, G. and Greenberg, D., Improved computational methods for ray tracing. *ACM Trans. Graph.* **3**(1), 52–69, January 1984.
64. Weiler, K. and Atherton, P., Hidden surface removal using polygon area sorting. *Comput. Graph.* **11**(2), 214–222, July 1977.
65. Whitted, T., An improved illumination model for shaded display. *Commun. ACM* **23**(6), 343–349, June 1980.
66. Wyvill, G. and Kunii, T.L., A functional model for constructive solid geometry. *The Visual Computer*, Vol. 1, No. 1, pp. 3–14, July 1985.
67. Wyvill, G., McPheeters, C. and Wyvill, B., Soft objects. *Advanced Computer Graphics* (Proc. of Computer Graphics Tokyo '86), pp. 113–128, April 1986.
68. Wyvill, G., Kunii, T.L. and Shiral, Y., Space division for ray tracing in CSG. *IEEE Comput., Graph. Appl.* **6**(4), 28–34, April 1986.
69. Yau, Mann-May and Srihari, S.N., A hierarchical data structure for multidimensional digital images. *Commun. ACM* **26**(7), 504–515, July 1983.
70. Youssef, S., Vectorized Simulation and Ray Tracing. Supercomputer Computations Research Institute, October 1987, FSU-SCRI-87-63.
71. Youssef, S., A new algorithm for object oriented ray tracing. *Comput. Vis. Graph. Image Process.* **34**, 125–137, 1986.