Comput. & Graphics Vol. 13, No. 4, pp. 445–460, 1989
Printed in Great Britain.

0097-8493/89 $3.00 + .00

Technical Section

IMPLEMENTING RAY TRACING WITH OCTREES AND NEIGHBOR FINDING*

HANAN SAMET
Computer Science Department, Institute of Advanced Computer Studies and Center for Automation Research, University of Maryland, College Park, MD 20742

Abstract—A ray tracing implementation is described that is based on an octree representation of a scene. Rays are traced through the scene by calculating the blocks through which they pass. This calculation is performed in a bottom-up manner through the use of neighbor finding. The octrees are assumed to be implemented by a pointer representation.

1. INTRODUCTION

The most basic operation in computer graphics is the conversion of an internal model of a three-dimensional scene into a two-dimensional scene that lies on the viewplane. The purpose is to generate an image of the scene as it would appear from a given viewpoint, and to display it on a two-dimensional screen. The situation becomes complex when we take into account the position of the light source, the presence of multiple light sources, and the possibility that light is reflected as well as refracted. This requires a calculation of what light falls on the object position represented by a pixel in the viewplane and is known as the *image rendering* task. Ray tracing[19] is an image rendering technique that models light as particles moving in the scene.

In this paper we focus on the use of hierarchical data structures such as the octree to speed up the determination of the objects that are intersected by rays emanating from the viewpoint. Our presentation is organized as follows. Section 2 contains some definitions and a description of our notation. Section 3 discusses the ray tracing task and shows how to trace a ray using neighbor finding in a scene represented by an octree. Section 4 gives a detailed implementation while Section 5 gives an example in two dimensions. Section 6 concludes with a brief discussion of some of the pitfalls of our solution, and those of alternative methods.

2. DEFINITIONS AND NOTATION

The *region octree*[8, 10, 15] is an extension of the quadtree data structure[14] to represent three-dimensional data (for a detailed discussion of such data structures, see [21–23, 26, 27]). We start with a $2^n \times 2^n \times 2^n$ object array of unit cubes (termed *voxels* or *obels*). The region octree is based on the successive subdivision of an object array into octants. If the array does not consist entirely of 1s or entirely of 0s, then it is subdivided into octants, suboctants, etc., until cubes (possibly single voxels) are obtained that consist of 1s or of 0s; *i.e.*, they are entirely contained in the region or entirely disjoint from it. This process is represented by a tree of degree 8 in which the root node represents the entire object, and the leaf nodes correspond to those cubes of the array for which no further subdivision is necessary. Leaf nodes are said to be black or white (alternatively, FULL or VOID) depending on whether their corresponding cubes are entirely within or outside of the object, respectively. All nonleaf nodes are said to be gray. Fig. 1a is an example of a simple three-dimensional object, in the form of a staircase, whose region octree block decomposition is given in Fig. 1b, and whose tree representation is given in Fig. 1c.

One of the problems with the region octree is that when the faces of the object are not orthogonal, the data structure requires much decomposition, and hence much space. In order to remedy this problem, a set of decomposition criteria is used such that no node contains more than one face, edge, or vertex unless the faces all meet at the same vertex or are adjacent to the same edge. We term the resulting structure a *PM octree* (see [1, 2, 4, 9, 16, 18, 28, 29]). For example, Fig. 2b is a PM octree decomposition of the object in Fig. 2a. This representation is quite useful since its space requirements for polyhedral objects are significantly smaller than those of a region octree. In two dimensions, we have a PM_1 quadtree which has the property that no node contains more than one edge or vertex unless the edges all meet at the same vertex[25]. For example, Fig. 3 is a PM_1 quadtree of a 5-sided polygon.

In order to understand the presentation of the algorithms, we first give some definitions and explain our notation. Fig. 4 shows the coordinate system that we are using relative to a cube. It is slightly different than the one used to generate Fig. 1. Let L and R denote the resulting lower and upper halves, respectively, when the x axis is partitioned. Let D and U denote the resulting lower and upper halves, respectively, when the y axis is partitioned. Let B and F denote the resulting lower and upper halves, respectively, when the z axis is partitioned. Fig. 5 illustrates the labelings corresponding to the partitions.

The labelings in Fig. 5 are also used to identify the faces, edges, and vertices of the cube as shown in Fig. 6. The faces are L (left), R (right), D (down), U (up), B (back), and F (front); however, only R, U, and F are visible. The edges and vertices of the cube are labeled by using an appropriate concatenation of labels of the adjacent faces. Note that vertex LDB and edges LD,

* This paper was originally planned to be published in the Special Issue, "3D Voxel-Based Graphics," with Arie Kaufman as Guest Editor, Vol. 13, No. 2 (1989).

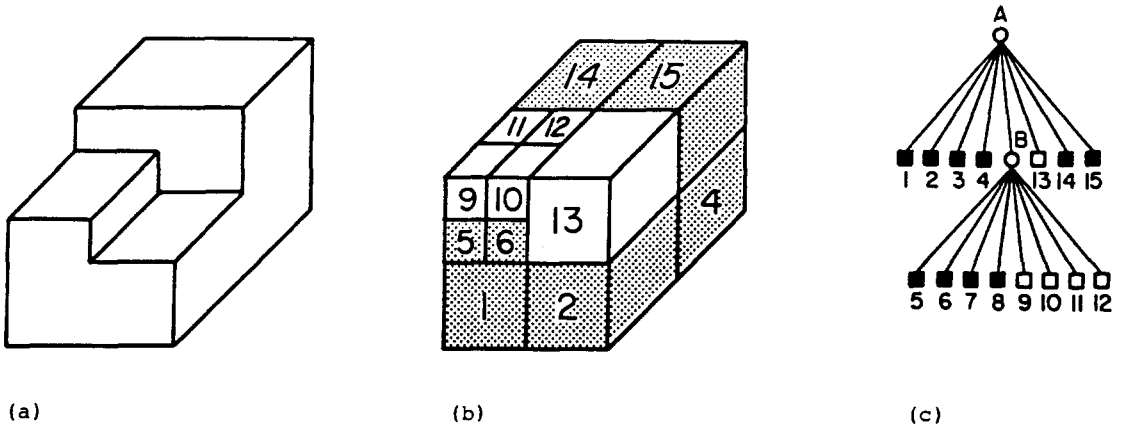

Fig. 1. (a) Example three-dimensional object, (b) its region octree block decomposition, and (c) its tree representation.

LB, and DB are not visible. Similarly, the octants are labeled by using a concatenation of these labels as shown in Fig. 7 (octant LDB is not visible). Fig. 8 is a numerical labeling for the octants (octant 0 is not visible).

The concept of a neighbor in an octree[24] is defined analogously to that in a quadtree[20]. We say that node Q is a *neighbor* of node R in direction I if Q corresponds to the smallest block (it may correspond to a nonleaf node) adjacent to R (*i.e.*, touching even if just at a point) in direction I of size greater than or equal to the block corresponding to R.

In two dimensions, there are 8 possible neighbor directions. In three dimensions, there are 26 possible directions. In particular, in two dimensions, two nodes can be adjacent, and hence neighbors, along an edge (4 possibilities) or along a vertex (4 possibilities). In contrast, in three dimensions, two nodes can be adjacent, and hence neighbors, along a face (6 possibilities), along an edge (12 possibilities), or along a vertex (8 possibilities). Such neighbors are termed face-neighbors, edge-neighbors, and vertex-neighbors, respectively. These relations are shown in Figs. 9a, 9b, and 9c, respectively.

We now describe an octree implementation that uses pointers. Assume that each octree node is stored as a record of type *node* containing 10 fields. The first nine fields contain pointers to the node's father and its eight sons, which correspond to the eight octants. If the node is a leaf node, then it will have eight pointers to the empty record. If P is a pointer to a node and O is an octant, then these fields are referenced as FATHER(P)

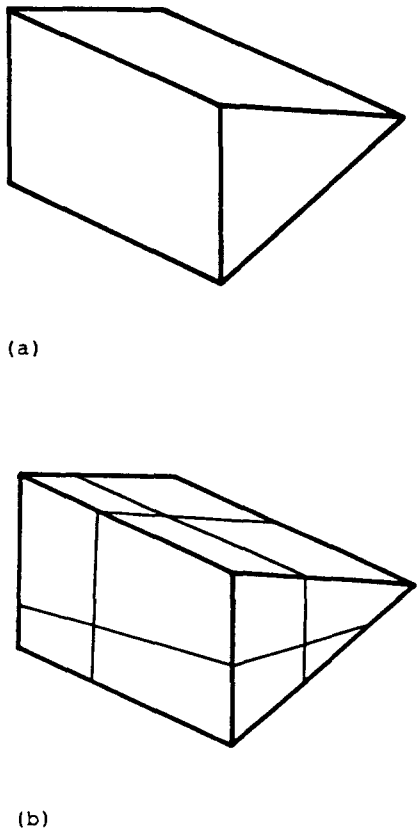

Fig. 2. (a) Example three-dimensional object, and (b) its corresponding PM octree.

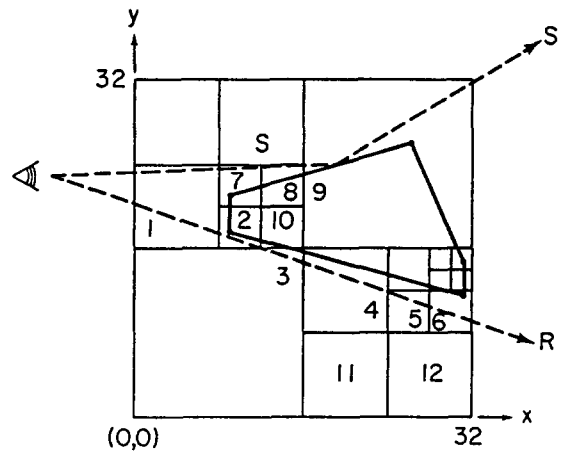

Fig. 3. PM_1 quadtree representation of a polygon. The cells intersected by rays R and S, emanating from the viewpoint, are labeled.

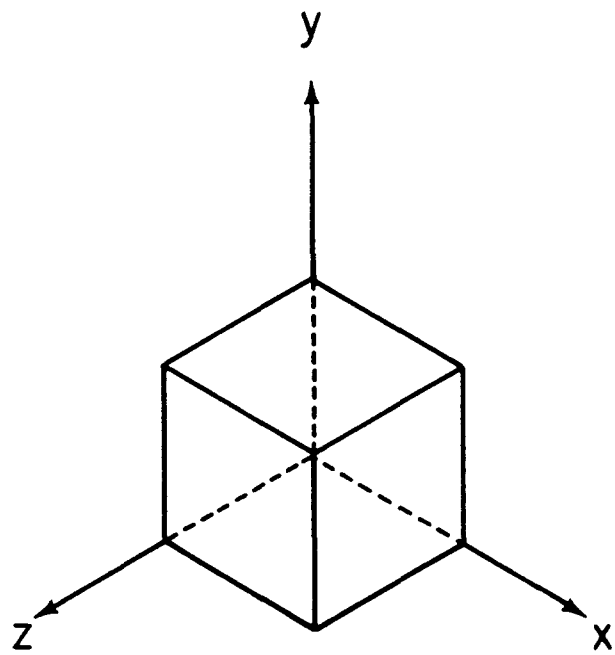

Fig. 4. Three-dimensional coordinate system.

and SON(P,O), respectively. We can determine the specific octant in which a node, say P, lies relative to its father by use of the function SONTYPE(P), which has a value of O if SON(FATHER(P),O) = P. The 10th field contains type information. For example, for a region octree it contains the color of the block of the image which the node represents, *i.e.*, black, white, or gray. The pointer from a node to its father is not required but is introduced here to ease the motion between arbitrary nodes in the octree.

3. THE RAY TRACING TASK

Ray tracing is an approximate simulation of how the light propagated through a scene lands on the image plane. This simulation is based on the geometric optics approach to reflection and refraction[30]. Although the geometry of the reflection and refraction of "particles" of light from surfaces is straightforward, the formulation of the equations to model the intensity of the light as it leaves these surfaces is a recent development. The quality of the displayed image is a function of the appropriateness of the model represented by these equations and the precision with which the scene is represented.

From a procedural point of view, the term *ray tracing* (or *ray casting*) describes the process of casting a ray from the viewpoint through a given pixel on the image plane that appears between the viewpoint and the scene. The pixel is identified with the surface of the nearest object that intersects the ray. This ray is termed the *primary ray*. If an intersection is found, then a number of additional rays are often cast. These additional rays are used to calculate shadows when there is another object between the light source and the intersected object, and to calculate indirect paths between the light source and the surface.

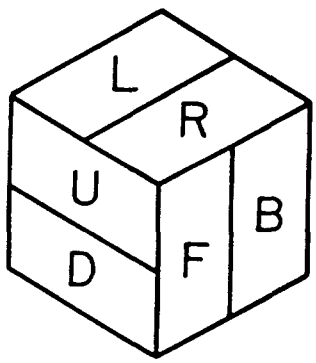

Fig. 5. Three orthogonal partitions of a cube.

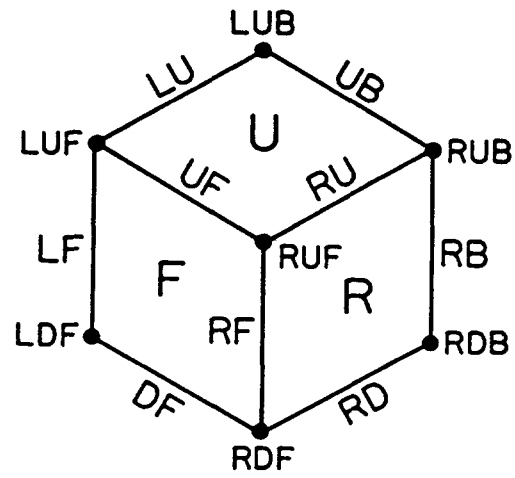

Fig. 6. Labeling of faces, edges, and vertices based on the partitioning defined in Fig. 5.

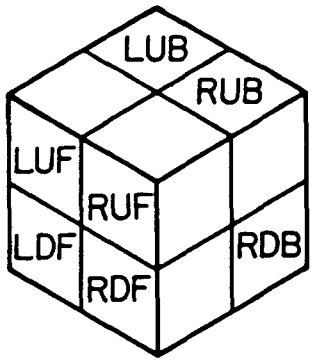

Fig. 7. Labeling of octants based on the partitioning defined in Fig. 5 (octant LDB is not visible).

When the object is opaque, then the additional rays are only used for modeling reflection and shadows. On the other hand, when we can see through the object, then the additional rays are also used for modeling transparence or translucence (in which case the ray may undergo refraction). These additional rays are termed *secondary rays*. Fig. 10 illustrates these terms. Although the distinction between reflected and refracted rays is interesting from a physical standpoint, this distinction has little effect from an algorithmic standpoint. In this paper, any reference to reflected rays is actually a reference to both reflected and refracted rays.

The amount of time required to display a scene is heavily influenced by the cost of tracing the path of the rays of light as they move backward from the viewer's eye, through the pixels of the image plane, and out through the scene. The motivation for using the octree in ray tracing is to enable the calculation of more rays with a greater amount of accuracy. Since light-modeling equations rely on the availability of accurate information about the location of the normal to the surface at the point of its intersection with the ray, PM octrees are generally more appropriate than region octrees. This is especially true for PM octrees that can represent curved, rather than planar, surfaces using either curved patches[17] or curved primitives[31].

Octrees have been used to speed up intersection calculations for ray tracing[5, 7, 11–13, 31]. The basic speedup can be seen by examining the PM_1 quadtree in Fig. 3. We use a quadtree instead of an octree in order to simplify the presentation. A naive ray tracing algorithm would have to test the ray emanating from the viewpoint against each of these sides, sort the resulting intersections, calculate the reflected ray, and finally test the reflected ray to see if it intersects any other portion of the polygon.

For example, consider ray S in Fig. 3 and assume that the boundary of the polygon is opaque (*i.e.*, no light is transmitted through it). Thus, the only secondary ray corresponds to reflection. From Fig. 3, we see that a quadtree-based algorithm would perform the calculation of ray S by visiting only 4 cells (*i.e.*, cells 1, 7, 8, and 9).

Once the scene's octree has been built (consisting of cells), we must trace each ray through it. We adopt the convention that for a ray to pass through a cell (as well as intersect an object), it must enter and exit the cell (or object) at two distinct points. Thus, a ray that is tangent to a cell (or object) at just one point does not pass through (intersect) the cell (object). On the other hand, a ray that is tangent to a cell (object) along an edge or a face of a cell (object) is said to pass through (intersect) the cell (object). For example, ray R in Fig. 3 passes directly from cell 2 to cell 3, without passing through cell 10. This convention is very important as otherwise an error may arise (see Section 6).

For each cell through which the ray passes, we only intersect the ray with the objects in that cell. If it intersects more than one object, then we determine the appropriate object and continue to trace the secondary rays, if necessary. If the ray does not intersect any of the objects in the cell, then we project the ray into the next cell and try again. As long as the cost of moving between adjacent cells is relatively low, we will save time over the cost of intersecting the ray with every object in the scene.

There are a number of methods of projecting the ray into succeeding blocks. Jansen[11] discusses these methods in a general manner. They can be best characterized as being either top-down or bottom-up. In this paper we focus on the bottom-up method. It follows the ray in the sense that first the closest bounding volume or cell, say C, to the viewpoint that is intersected by the ray is located. Let P be the point at which the ray leaves C. If C does not contain an object that intersects the ray, then locate the smallest cell or bounding volume, say C', that contains the point $Q = P + \Delta$. There are many methods of locating C'. One possibility is to perform a point location algorithm which starts at the root of the tree. Another variation, and the one we describe, is to use neighbor-finding methods[20, 24].

Assume that the ray is defined parametrically by

$$x = m_x \cdot t + b_x, \tag{1}$$

$$y = m_y \cdot t + b_y, \tag{2}$$

$$z = m_z \cdot t + b_z. \tag{3}$$

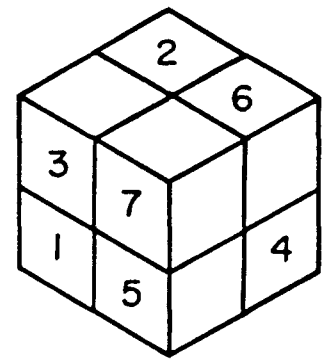

Fig. 8. Numeric labeling of octants based on the partitioning defined in Fig. 5 (octant 0 is not visible).

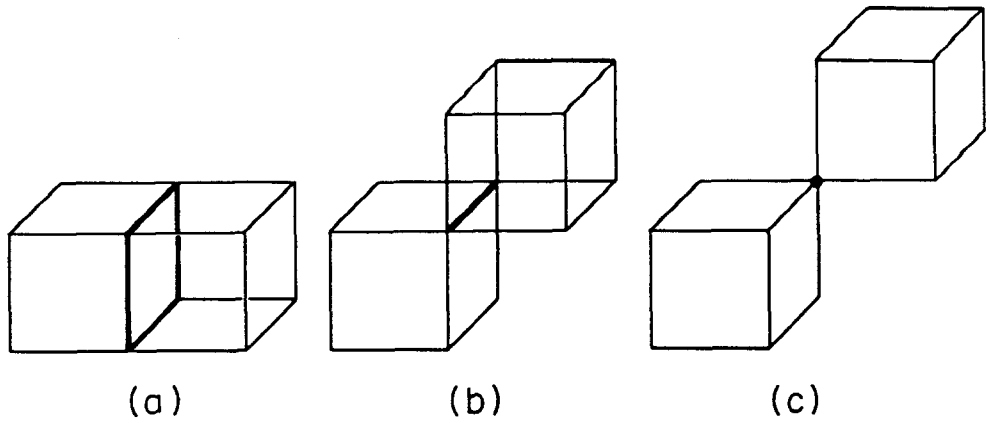

Fig. 9. Example of (a) a face neighbor, (b) an edge neighbor, and (c) a vertex neighbor.

One way to determine the parameters is to choose two points on the ray and let one correspond to $t = 0$ and the other to $t = 1$, and then to solve the six equations. The ray tracing computation is simplified when the parameters are integers. This situation is assured when $t = 0$ corresponds to the viewpoint, and when the viewpoint and the point corresponding to $t = 1$ both have integer coordinates. Note that $t \geq 0$ for every point on the ray. We also assume that the origin of the three-dimensional space containing the scene is at (0, 0, 0) and the width of the space is a power of 2. The smallest possible cell is of width 1.

In practice, the situation is not so simple. In particular, when the viewplane is in an arbitrary position in space, it is usually not the case that every pixel on the viewplane has integer coordinates relative to the viewpoint. Nevertheless, we do know that (b_x, b_y, b_z) are equal to the coordinate values of the viewpoint. In the following, we describe a more general solution which permits the m_i to be rational numbers while assuming that the viewpoint has integer coordinates. Note that our solution can be modified to permit the viewpoint to have rational coordinates, but this is not done here.

Let $B = (b_x, b_y, b_z)$ be the viewpoint. Assume that the viewplane is defined by the three points Q, R, and S such that $Q = (q_x, q_y, q_z)$ is the origin of the viewplane. Let $\vec{J}$ and $\vec{K}$ be the base vectors in the viewplane. Assume that $\vec{J} = j_x\vec{\alpha}_x + j_y\vec{\alpha}_y + j_z\vec{\alpha}_z$ and $\vec{K} = k_x\vec{\alpha}_x + k_y\vec{\alpha}_y + k_z\vec{\alpha}_z$, where j_i and k_i are rational numbers, and $\vec{\alpha}_i$ are unit vectors in the x, y, and z directions. Note that $\vec{J}$ and $\vec{K}$ are base vectors, although $j_x^2 + j_y^2 + j_z^2$ and $k_x^2 + k_y^2 + k_z^2$ do not necessarily equal 1. A point $P(u, v)$ on the viewplane (u and v are viewplane coordinates) can be written as:

$$P(\vec{u, v}) = u\vec{J} + v\vec{K} + \vec{Q}.$$

Ray R from the viewpoint B through point $P(u, v)$ on the viewplane can be expressed as:

$$\vec{R} = (\vec{P} - \vec{B})t + \vec{B}.$$

Expanding this equation yields:

$$x = (j_x \cdot u + k_x \cdot v + q_x - b_x) \cdot t + b_x, \qquad (4)$$

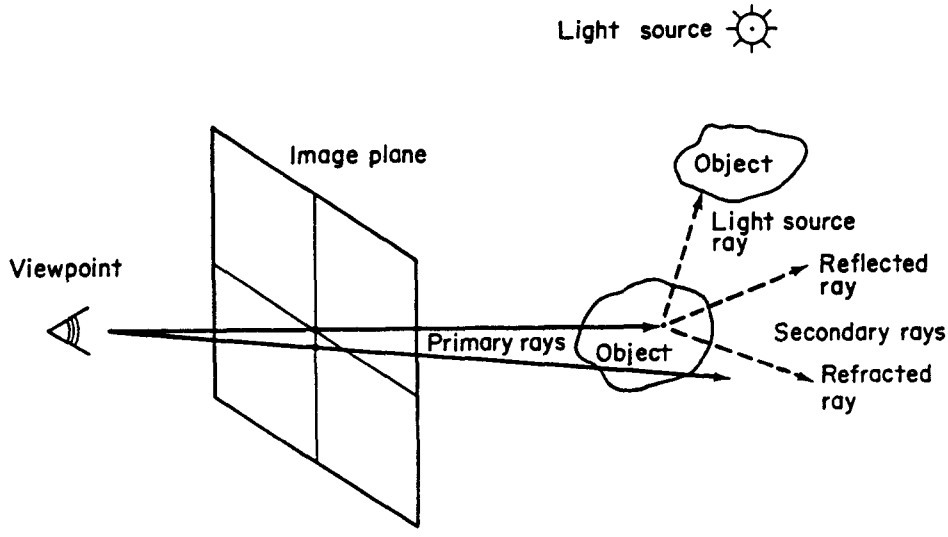

Fig. 10. Illustration of ray tracing. Solid lines correspond to primary rays while broken lines correspond to secondary and light source rays.

$$y = (j_y \cdot u + k_y \cdot v + q_y - b_y) \cdot t + b_y, \quad (5)$$

$$z = (j_z \cdot u + k_z \cdot v + q_z - b_z) \cdot t + b_z. \quad (6)$$

The coefficients of t in Eqns. (4)–(6) correspond to the values of m_i and are rational numbers. In fact, as can be seen below, by redefining the parametric equations for the ray in terms of the lowest common denominator of m_x, m_y, and m_z, say, c, all of the parameters are integers. In the following, $t' = t/c$, and the values of m_i' are the remaining numerators once the denominators have been set to c.

$$x = m_x' \cdot t' + b_x, \quad (7)$$

$$y = m_y' \cdot t' + b_y, \quad (8)$$

$$z = m_z' \cdot t' + b_z. \quad (9)$$

In the remainder of the discussion, we assume that the parametric equations have been manipulated in such a manner. We shall use m_i and t, although we are actually referring to m_i' and t', respectively.

Tracing the ray is achieved by the following three-step process. First, we must show how to compute the points at which the ray enters and exits the cell (*i.e.*, clip the ray). This process is a simplification of the Cyrus-Beck clipping algorithm[3, 19] and is the one used by Glassner[7], as well as by Wyvill and Kunii[31]. Glassner does not describe an implementation. Wyvill and Kunii's implementation is discussed in Section 6.

The nature of the implementation is very important and requires much care since the computation must be exact. In particular, we cannot use floating point arithmetic. Instead, we use rational arithmetic. Next, we process the cell by intersecting the ray with the objects in the cell. Finally, if necessary (*i.e.*, the ray does not intersect any of the objects in the cell), we compute the direction of the next cell intersected by the ray and also locate it.

To determine the points at which a ray enters and exits a three-dimensional cell, each of whose sides is of width W, we test the ray against the bounding planes (*i.e.*, faces) of the volume corresponding to the cell. For example, consider a cell bounded by $x = x_0$ and $x_0 + W$, $y = y_0$ and $y_0 + W$, and $z = z_0$ and $z_0 + W$. We compute a value of t for each of $x = x_0$, $x = x_0 + W$, $y = y_0$, $y = y_0 + W$, $z = z_0$, and $z = z_0 + W$.

Let t_i^{in} and t_i^{out} correspond to the range of values of t taken by coordinate i. In particular, if $m_i < 0$, then t_i^{in} and t_i^{out} correspond to $i = i_0 + W$ and $i = i_0$, respectively, whereas if $m_i \geq 0$, then t_i^{in} and t_i^{out} correspond to $i = i_0$ and $i = i_0 + W$, respectively. The intersection of these three ranges of t yields the values that the ray may assume while it is in the cell. In particular, t will range between $\max(t_i^{in})$ and $\min(t_i^{out})$.

To process the next cell C', we must locate it. This requires us first to determine its direction, say I, relative to the current cell C. The computation of I is a critical part of the location process and cannot be ignored (see Section 6). The direction depends on the location of the point, say P, at which the ray exits C. We have three possible positions—P is either on a vertex, edge, or face of C. P is on a vertex if t_i^{out} has the same value for each coordinate i. P is on an edge if t_i^{out} has the same value for two of the coordinates i. Otherwise, P is on a face (*i.e.*, t_i^{out} has a different value for each coordinate i).

Since the values of t are not necessarily integers, and as we need to perform a test involving equality (not within a tolerance!), we represent t as a rational number (*i.e.*, an ordered pair consisting of a numerator and a denominator). Comparisons involving different values of t are made by cross-multiplying the numerators and denominators of the comparands and comparing the results.

Now that we know the direction of C' with respect to C, we must locate it. We have two alternatives. The first alternative is to use a point location algorithm. We compute a point, say Q, that is guaranteed to be in C'. Finding the cell containing point Q is easy. We start at the root of the octree, say G, and descend it based on a comparison of Q with the center of the block corresponding to G. The descent ceases once we reach a leaf node. This approach is commonly used [13, 31].

The computation of Q is relatively straightforward, although its implementation requires us to pay close attention to details. Q depends on the location of P, the point at which the ray exits C. Let $P = (P_x, P_y, P_z)$ and $Q = (Q_x, Q_y, Q_z)$. Let $I = (I_x, I_y, I_z)$ be the direction of the next cell C'. We follow the convention that the left, down, and back faces of a cell are closed, *i.e.*, if a point lies on one of these faces, then it is in the cell bounded by them. To calculate Q using this convention we subtract Δ (where Δ is very small) from P_j if I_j is in the negative (*i.e.*, decreasing) direction of j. Δ must be no larger than the width of the smallest possible cell, *i.e.*, 1. Δ cannot be smaller than 1 because we are using integer arithmetic in the process of locating the cell containing Q.

For example, if I = 'LUB', then we must subtract one from P_x and P_z with the result that $Q = (P_x - 1, P_y, P_z - 1)$. On the other hand, when I = 'RD', we need only subtract one from P_y while the remaining values remain the same, *i.e.*, $Q = (P_x, P_y - 1, P_z)$. Note that our conventions with respect to which faces are closed enable us to use the integer parts of the coordinate values that are not in the I direction. Thus, when I = 'RD', we use the integer parts of P_z. P_x and P_y are already integers by virtue of being on the edge of a cell.

The second alternative, and the one we use, makes use of neighbor-finding methods[20, 24]. In particular, we find the neighbor of C, say N, in direction I having a width which is greater than or equal to that of C. If such a neighbor does not exist, then we are at the border of the three-dimensional space, and we exit. If N does not correspond to a gray node, then we are done (*i.e.*, $C' = N$). Otherwise, we now calculate a point Q that is guaranteed to be in C' which is a descendant of N (recall that N's node is gray). We locate C' by applying the point location algorithm described above.

The advantage of this approach over just using the point location algorithm is that fewer nodes will be visited since we need not descend from the root of the tree. Also, traversing links in the octree by using neighbor finding is considerably cheaper than the arithmetic operations that are part of the point location algorithm.

When the octree is represented using pointers, then neighbor finding is implemented by using the FATHER links. On the other hand, a pointer-less octree representation can also be used[6]. One example is as a collection of the leaf nodes comprising the octree where each leaf node, say P, is represented by a pair of numbers known as its locational code. The first number is the depth of the tree at which P is located. The second number is formed by concatenating the base 8 digits corresponding to directional codes that locate P along a path from the root of the octree. In such a case, a neighboring node is located by first manipulating the bits that comprise the second number corresponding to P based on the direction of the desired neighbor, and then performing a search.

4. SAMPLE IMPLEMENTATION

An implementation of the bottom-up process of tracing a ray through a scene, represented by an octree that uses neighbor finding to locate successive cells, is given by the following procedures. The process is controlled by procedure RAY_TRACER. It is invoked with parameters corresponding to the parametric representation of the traced ray, a pointer to the root of the octree, and the width of the scene.

RAY_TRACER's first action is to determine the value of t, if any, for the point, given by POINT, at which the ray first enters the cell corresponding to the entire scene and the direction of the ray relative to the face, edge, or vertex containing POINT. This is achieved by procedure FIRST_POINT. If POINT lies outside of the scene, then the process stops since there are no intersections. Otherwise, the particular cell containing POINT is located by use of procedure FIND_3D_BLOCK. The function OFFSET, given in Table 1, contains multiplicative factors that facilitate the calculation of the coordinate values of the furthest corners of the sons of each node in procedure FIND_3D_BLOCK. In particular, OFFSET(A,O) is the multiplicative factor for the calculation of the value of coordinate A when descending to the son in octant O.

Once the first cell intersected by the ray has been located, the ray is traced through successive cells. For each cell through which the ray passes, a record of type *cell* is created that has 6 fields called T_IN, T_OUT, SIZ, PTR, CORNER, and DIRECT. Letting C be a pointer to a record of type *cell*, T_IN(C) and T_OUT(C) indicate the values of t for the points at which the ray enters and exits from C. SIZ(C) is the width of C's cell. PTR(C) is a pointer to C's node in the octree. CORNER(C)[I] is the value of the Ith coordinate of cell C's furthest corner from the origin. DIRECT(C) is the direction of the next cell, relative to cell C, through which the ray must be traced. Procedure RAY_INTERSECTS_OBJECT_IN_CELL, not given here, performs the actual intersection tests of the ray with the objects associated with cell C.

If procedure RAY_INTERSECTS_OBJECT_IN_CELL determines that the ray intersects the object, then a reflection or refraction calculation must be made. This is equivalent to tracing a new ray and is not in the code given here, although it is discussed below. Otherwise, the ray is traced into the next cell. This cell is determined by use of neighbor finding via a call to procedure OT_GTEQ_NEIGHBOR that returns a pointer P. OT_GTEQ_NEIGHBOR is aided by the function TYPE to determine the type of the neighbor's direction (*i.e.*, face, edge, or vertex) so that it can invoke the appropriate neighbor-finding routine (OT_GTEQ_FACE_NEIGHBOR2, OT_TEQ_EDGE_NEIGHBOR2, or OT_GTEQ_VERTEX_NEIGHBOR2).

The code for the neighbor finding procedures makes use of the predicate ADJ, and the functions REFLECT, COMMON_FACE, and COMMON_EDGE to aid in the expression of operations involving a block's octants and its faces, edges, and vertices. ADJ(I,O) is true if and only if octant O is adjacent to the Ith face, edge, or vertex of O's containing block. REFLECT(I,O) yields the SONTYPE value of the block of equal size (not necessarily a brother) that shares the Ith face, edge, or vertex of a block having SONTYPE value O. COMMON_FACE(I,O) yields the type of the face (*i.e.*, label) of O's containing block that is common to octant O and its neighbor in the Ith direction (I is an edge or a vertex). COMMON_EDGE(I,O) yields the type of the edge (*i.e.*, label) of O's containing block that is common to octant O and its neighbor in the Ith direction (I is a vertex). Tables 2–5 contain their definitions. Ω denotes an undefined value.

If the cell pointed at by P does not correspond to a leaf node, then the point at which the ray first enters the next cell is calculated and FIND_3D_BLOCK is used to locate it, starting at P. The entire process stops

Table 1. OFFSET(A,O).

A (axis)	O (octant)							
	LDB	LDF	LUB	LUF	RDB	RDF	RUB	RUF
X	1	1	1	1	0	0	0	0
Y	1	1	0	0	1	1	0	0
Z	1	0	1	0	1	0	1	0

Table 2. ADJ(I,O).

I (direction)	O (octant)							
	LDB	LDF	LUB	LUF	RDB	RDF	RUB	RUF
L	T	T	T	T	F	F	F	F
R	F	F	F	F	T	T	T	T
D	T	T	F	F	T	T	F	F
U	F	F	T	T	F	F	T	T
B	T	F	T	F	T	F	T	F
F	F	T	F	T	F	T	F	T
LD	T	T	F	F	F	F	F	F
LU	F	F	T	T	F	F	F	F
LB	T	F	T	F	F	F	F	F
LF	F	T	F	T	F	F	F	F
RD	F	F	F	F	T	T	F	F
RU	F	F	F	F	F	F	T	T
RB	F	F	F	F	T	F	T	F
RF	F	F	F	F	F	T	F	T
DB	T	F	F	F	T	F	F	F
DF	F	T	F	F	F	T	F	F
UB	F	F	T	F	F	F	T	F
UF	F	F	F	T	F	F	F	T
LDB	T	F	F	F	F	F	F	F
LDF	F	T	F	F	F	F	F	F
LUB	F	F	T	F	F	F	F	F
LUF	F	F	F	T	F	F	F	F
RDB	F	F	F	F	T	F	F	F
RDF	F	F	F	F	F	T	F	F
RUB	F	F	F	F	F	F	T	F
RUF	F	F	F	F	F	F	F	T

Table 3. REFLECT(I,O).

I (direction)	O (octant)							
	LDB	LDF	LUB	LUF	RDB	RDF	RUB	RUF
L	RDB	RDF	RUB	RUF	LDB	LDF	LUB	LUF
R	RDB	RDF	RUB	RUF	LDB	LDF	LUB	LUF
D	LUB	LUF	LDB	LDF	RUB	RUF	RDB	RDF
U	LUB	LUF	LDB	LDF	RUB	RUF	RDB	RDF
B	LDF	LDB	LUF	LUB	RDF	RDB	RUF	RUB
F	LDF	LDB	LUF	LUB	RDF	RDB	RUF	RUB
LD	RUB	RUF	RDB	RDF	LUB	LUF	LDB	LDF
LU	RUB	RUF	RDB	RDF	LUB	LUF	LDB	LDF
LB	RDF	RDB	RUF	RUB	LDF	LDB	LUF	LUB
LF	RDF	RDB	RUF	RUB	LDF	LDB	LUF	LUB
RD	RUB	RUF	RDB	RDF	LUB	LUF	LDB	LDF
RU	RUB	RUF	RDB	RDF	LUB	LUF	LDB	LDF
RB	RDF	RDB	RUF	RUB	LDF	LDB	LUF	LUB
RF	RDF	RDB	RUF	RUB	LDF	LDB	LUF	LUB
DB	LUF	LUB	LDF	LDB	RUF	RUB	RDF	RDB
DF	LUF	LUB	LDF	LDB	RUF	RUB	RDF	RDB
UB	LUF	LUB	LDF	LDB	RUF	RUB	RDF	RDB
UF	LUF	LUB	LDF	LDB	RUF	RUB	RDF	RDB
LDB	RUF	RUB	RDF	RDB	LUF	LUB	LDF	LDB
LDF	RUF	RUB	RDF	RDB	LUF	LUB	LDF	LDB
LUB	RUF	RUB	RDF	RDB	LUF	LUB	LDF	LDB
LUF	RUF	RUB	RDF	RDB	LUF	LUB	LDF	LDB
RDB	RUF	RUB	RDF	RDB	LUF	LUB	LDF	LDB
RDF	RUF	RUB	RDF	RDB	LUF	LUB	LDF	LDB
RUB	RUF	RUB	RDF	RDB	LUF	LUB	LDF	LDB
RUF	RUF	RUB	RDF	RDB	LUF	LUB	LDF	LDB

Table 4. COMMON_FACE(I,O).

I (direction)	O (octant)							
	LDB	LDF	LUB	LUF	RDB	RDF	RUB	RUF
LD	Ω	Ω	L	L	D	D	Ω	Ω
LU	L	L	Ω	Ω	Ω	Ω	U	U
LB	Ω	L	Ω	L	B	Ω	B	Ω
LF	L	Ω	L	Ω	Ω	F	Ω	F
RD	D	D	Ω	Ω	Ω	Ω	R	R
RU	Ω	Ω	U	U	R	R	Ω	Ω
RB	B	Ω	B	Ω	Ω	R	Ω	R
RF	Ω	F	Ω	F	R	Ω	R	Ω
DB	Ω	D	B	Ω	Ω	D	B	Ω
DF	D	Ω	Ω	F	D	Ω	Ω	F
UB	B	Ω	Ω	U	B	Ω	Ω	U
UF	Ω	F	U	Ω	Ω	F	U	Ω
LDB	Ω	Ω	Ω	L	Ω	D	B	Ω
LDF	Ω	Ω	L	Ω	D	Ω	Ω	F
LUB	Ω	L	Ω	Ω	B	Ω	Ω	U
LUF	L	Ω	Ω	Ω	Ω	F	U	Ω
RDB	Ω	D	B	Ω	Ω	Ω	Ω	R
RDF	D	Ω	Ω	F	Ω	Ω	R	Ω
RUB	B	Ω	Ω	U	Ω	R	Ω	Ω
RUF	Ω	F	U	Ω	R	Ω	Ω	Ω

when either a ray intersects an object within a cell or the ray exits the scene (*i.e.*, OT_GTEQ_NEIGHBOR returns a pointer to NIL).

To be able to compare different values of $\vec{t}$ so that the direction of the next cell can be determined, we need to compute the minimum and maximum values of t. This must be done in an exact manner and, thus, we represent the values of t as rational numbers by use of a record of type *rational*, with two fields NUM and DEN corresponding to the numerator and denominator, respectively.

The actual comparisons are aided by using procedure COMPARE_T to precompute pairwise comparisons, *i.e.*, CYX, CZX, and CZY. These comparisons are used by procedure NEXT_CELL_DIRECTION to determine the direction of the next cell, relative to the present cell, that is intersected by the ray. This is facilitated by making use of the sign of M and functions FACE_DIR, EDGE_DIR, and VERTEX_DIR given in Tables 6, 7, and 8, respectively.

At times, we need to calculate the coordinates of a point in a specific cell. This situation arises when attempting to locate the first cell that is intersected by the ray, when attempting to locate a neighboring cell that is smaller than the current cell, and when setting the CORNER field of a record of type *cell*. The function CHANGE(I,A) facilitates this task by indicating the smallest amount, with the appropriate sign, by which the value of coordinate A changes due to motion in direction I. For example, CHANGE('RB', 'Z') = −1 as the value of coordinate z will decrease as a result of motion in direction 'RB'. On the other hand, CHANGE('RB', 'Y') = 0, as the value of coordinate y is unaffected by motion in direction 'RB'. CHANGE is given in Table 9.

As stated above, to handle reflection and refraction at a surface properly, we need to trace the appropriate ray anew. This can be done in the same manner starting at the point at which the primary ray intersects the surface. The secondary (*i.e.*, reflected and refracted) ray is also defined parametrically. The only difficulty is that the definition of the secondary ray will require a larger computer word size to cope with the increase in the number of binary digits necessary to specify the

Table 5. COMMON_EDGE(I,O).

I (direction)	O (octant)							
	LDB	LDF	LUB	LUF	RDB	RDF	RUB	RUF
LDB	Ω	LD	LB	Ω	DB	Ω	Ω	Ω
LDF	LD	Ω	Ω	LF	Ω	DF	Ω	Ω
LUB	LB	Ω	Ω	LU	Ω	Ω	UB	Ω
LUF	Ω	LF	LU	Ω	Ω	Ω	Ω	UF
RDB	DB	Ω	Ω	Ω	Ω	RD	RB	Ω
RDF	Ω	DF	Ω	Ω	RD	Ω	Ω	RF
RUB	Ω	Ω	UB	Ω	RB	Ω	Ω	RU
RUF	Ω	Ω	Ω	UF	Ω	RF	RU	Ω

parameters and values of t correctly. This is a direct result of the use of rational arithmetic. It can be avoided in part by using parametric equations in the form of (4)–(6) instead of (7)–(9). However, this requires that m be treated as data of type *rational* in procedure RAY_TRACER.

```
procedure RAY_TRACER(M,B,W,R);
/* Trace a ray given parametrically by M and B, i.e., x = m_x·t + b_x, through the octree rooted at R which corresponds to
   the three-dimensional space of width W with origin at (0, 0, 0). W is a power of 2. Procedure RAY_INTER-
   SECTS_OBJECT_IN_CELL performs object tests in each cell through which the ray passes. It is not given here. Its
   argument is a pointer to a record of type cell which has fields T_IN, T_OUT, SIZ, PTR, CORNER, and DIRECT
   corresponding to the value of the parameter t for the entry and exit points, the width of the cell, a pointer to its node in
   the octree, the coordinates of its furthest corner (from the origin), and the direction of the next cell through which the ray
   must be traced, respectively.*/
begin
  global integer array M,B[{'X','Y','Z'}];
  value integer W;
  value pointer node R;
  global integer array SIGN_M[{'X','Y','Z'}];
  pointer cell C;
  rational pointer array T[{'X','Y','Z'}];
  integer array POINT[{'X','Y','Z'}];
  direction DIR;
  integer CYX,CZX,CZY;
  axis I,MIN_AXIS;
  pointer node P;
  P ← R;
C ← create(cell);
for I in {'X','Y','Z'} do /* Keep track of the direction of the ray */
  SIGN_M[I] ← if M[I] > 0 then 0
              else 1;
/* Find t for the first entry point of the ray: */
T_OUT(C) ← FIRST_POINT(W,DIR);
for I in {'X','Y','Z'} do
  begin /* Calculate the first entry point */
    CORNER(C)[I] ← W;
    POINT[I] ← ⌊(M[I]*NUM(T_OUT(C)))/DEN(T_OUT(C))⌋ + B[I]
               + SIGN_M[I]*CHANGE(DIR,I); /* ⌊x⌋ is the floor of x */
  end;
if POINT['X'] < 0 or POINT['Y'] < 0 or POINT['Z'] < 0 or
  POINT['X'] ≥ W or POINT['Y'] ≥ W or POINT['Z'] ≥ W
  then return /* The ray never enters the space */
else /* Locate the closest cell to the entry point */
  FIND_3D_BLOCK(P,POINT,CORNER(C),W);
while true do /* Follow the ray through the space */
  begin
    PTR(C) ← P;
    SIZ(C) ← W;
    T_IN(C) ← T_OUT(C);
    for I in {'X','Y','Z'} do
      begin /* Compute a t value for the exit point for each plane */
        NUM(T[I]) ← CORNER(C)[I] − SIGN_M[I]*W − B[I];
        DEN(T[I]) ← M[I];
      end;
    COMPARE_T(T,CYX,CZX,CZY);
    /* Find the minimum of the values of t using rational arithmetic: */
    MIN_AXIS ← if CZY < 0 then
                 if CZX < 0 then 'Z'
                 else 'X'
               else if CYX < 0 then 'Y'
               else 'X';
DIRECT(C) ← NEXT_CELL_DIRECTION(MIN_AXIS,CYX,CZX,CZY);
T_OUT(C) ← T[MIN_AXIS];
if RAY_INTERSECTS_OBJECT_IN_CELL(C) then return
else
  begin
    /* Locate the next cell in direction DIRECT(C) using neighbor finding: */
    OT_GTEQ_NEIGHBOR(PTR(C),DIRECT(C),P,W);
    if null(P) then return; /* Neighbor does not exist */
    for I in {'X','Y','Z'} do /* Compute location of next cell */
      CORNER(C)[I] ← CORNER(C)[I]
                     + if CHANGE(DIRECT(C),I) = 1 then W
                       else if CHANGE(DIRECT(C),I) = −1 then −SIZ(C)
                       else if (CORNER(C)[I] mod W) = 0 then 0
                       else W − (CORNER(C)[I] mod W);
```

```
      if GRAY(P) then /* Neighbor is smaller */
        begin /* Compute a point within the neighbor */
          for I in {'X','Y','Z'} do /* ⌊x⌋ is the floor of x */
            POINT[I] ← ⌊(M[I]*NUM(T_OUT(C)))/DEN(T_OUT(C))⌋ + B[I]
                       + SIGN_M[I]*CHANGE(DIRECT(C),I);
          FIND_3D_BLOCK(P,POINT,CORNER(C),W); /* Locate cell */
        end;
      end;
    end;
end;

pointer rational procedure FIRST_POINT(W,DIR);
/* Return a pointer to a record containing the value of the ray parameter t corresponding to the point at which the ray first
   enters the three-dimensional space of width W through which the ray is traced. DIR is set to the direction of the ray. */
begin
  value integer W;
  reference direction DIR;
  rational pointer array T[{'X','Y','Z'}];
  global integer array M,B,SIGN_M[{'X','Y','Z'}];
  integer CYX,CZX,CZY;
  axis I,MAX_AXIS;
  for I in {'X','Y','Z'} do
    begin /* Compute a t value for the entry point for each plane */
      NUM(T[I]) ← SIGN_M[I]*W - B[I];
      DEN(T[I]) ← M[I];
    end;
  COMPARE_T(T,CYX,CZX,CZY);
  /* Find the maximum of the values of t using rational arithmetic: */
  MAX_AXIS ← if CZY > 0 then
                 if CZX > 0 then 'Z'
                 else 'X'
               else if CYX > 0 then 'Y'
               else 'X';
  DIR ← NEXT_CELL_DIRECTION(MAX_AXIS,CYX,CZX,CZY);
  return(T[MAX_AXIS]);
end;

procedure COMPARE_T(T,CYX,CZX,CZY);
/* Compute CYX, CZX, and CZY. CYX is the pairwise comparison of T['Y'] and T['X'], CZX is the pairwise comparison
   of T['Z'] and T['X'], and CZY is the pairwise comparison of T['Z'] and T['Y']. */
begin
  value rational point array T[{'X','Y','Z'}];
  reference integer CYX,CZX,CZY;
  CYX ← abs(NUM(T['Y'])*DEN(T['X'])) - abs(NUM(T['X'])*DEN(T['Y']));
  CZX ← abs(NUM(T['Z'])*DEN(T['X'])) - abs(NUM(T['X'])*DEN(T[Z']));
  CZY ← abs(NUM(T['Z'])*DEN(T['Y'])) - abs(NUM(T['Y'])*DEN(T['Z']));
end;

direction procedure NEXT_CELL_DIRECTION(A,CYX,CZX,CZY);
/* Return the direction of the next cell through which the ray must be traced. A is the axis corresponding to the value of t.
   CYX, CZX, and CZY are pairwise comparisons of the values of t for the bounding sides of the cell through which the ray
   is exiting. */
begin
  value axis A;
  value integer CYX,CZX,CZY;
  global integer array SIGN_M[{'X','Y','Z'}];
  return(if A = 'Z' then FACE_DIR('Z',SIGN_M['Z'])
         else if A = 'Y' then
           if CZY = 0 then EDGE_DIR('YZ',SIGN_M['Y'],SIGN_M['Z])
           else FACE_DIR('Y',SIGN_M['Y])
         else if CZX = 0 then
           if CYX = 0 then VERTEX_DIR(SIGN_M['X],SIGN_M['Y],SIGN_M['Z])
           else EDGE_DIR('XZ',SIGN_M['X],SIGN_M['Z])
         else if CYX = 0 then EDGE_DIR('XY',SIGN_M['X],SIGN_M['Y])
         else FACE_DIR('X',SIGN_M['X]));
end;

procedure FIND_3D_BLOCK(P,POINT,FAR,W);
/* P points to a node corresponding to a block of width W having its furthest corner from the origin at FAR (i.e., FAR['X'],
   FAR['Y'], and FAR['Z']). Find the smallest block in P containing the voxel whose nearest corner to the origin is at POINT.
   If P is black or white, then return the values of P, W, and FAR; otherwise, repeat the procedure for the son of P that
   contains POINT. */
begin
  reference point node P;
  value integer array POINT[{'X','Y','Z'}];
```

```
  reference integer array FAR[{'X','Y','Z'}];
  reference integer W;
  axis I;
  octant Q;
  while GRAY(P) do
    begin
      W ← W/2;
      Q ← GET_OCTANT(POINT['X'],FAR['X'] - W,
                     POINT['Y'],FAR['Y'] - W,
                     POINT['Z'],FAR['Z'] - W);
      for I in {'X','Y','Z'} do FAR[I] ← FAR[I] - OFFSET(I,Q)*W;
      P ← SON(P,Q);
    end;
end;

octant procedure GET_OCTANT(X,XCENTER,Y,YCENTER,Z,ZCENTER);
/* Find the octant of the block rooted at (XCENTER,YCENTER,ZCENTER) that contains (X,Y,Z). */
begin
  value integer X,XCENTER,Y,YCENTER,Z,ZCENTER;
  return (if X < XCENTER then
            if Y < YCENTER then
              if Z < ZCENTER then 'LDB'
              else 'LDF'
            else if Z < ZCENTER then 'LUB'
            else 'LUF'
          else if Y < YCENTER then
            if Z < ZCENTER then 'RDB'
            else 'RDF'
          else if Z < ZCENTER then 'RUB'
          else 'RUF');
end;

procedure OT_GTEQ_NEIGHBOR(P,D,Q,W);
/* Determine the type of direction D and invoke the appropriate neighbor-finding procedure. Q will contain the neighbor of
   node P, of size greater than or equal to P, in direction D. W denotes the length of a side of node P and the length of a side
   of node Q. If a neighboring node does not exist, then return NIL. */
begin
  value pointer node P;
  value direction D;
  reference pointer node Q;
  reference integer W;
  if TYPE(D) = 'FACE' then OT_GTEQ_FACE_NEIGHBOR2(P,D,Q,LOG2(W))
    /* LOG2 returns the base 2 log of W */
  else if TYPE(D) = 'EDGE' then
    OT_GTEQ_EDGE_NEIGHBOR2(P,D,Q,LOG2(W))
  else OT_GTEQ_VERTEX_NEIGHBOR2(P,D,Q,LOG2(W));
end;

recursive pointer node procedure OT_GTEQ_FACE_NEIGHBOR2(P,I,Q,L);
/* Return in Q the face-neighbor of node P, of size greater than or equal to P, in direction I. L denotes the level of the tree
   at which node P is initially found, and the level of the tree at which node Q is ultimately found. If such a node does not
   exist, then return NIL. For an octree corresponding to a 2^n × 2^n × 2^n image array, the root is at level n and a node at level
   i is at a distance of n - i from the root of the tree. */
begin
  value pointer node P;
  value face I;
  reference pointer node Q;
  reference integer L;
  L ← L + 1;
  if not(null(FATHER(P))) and ADJ(I,SONTYPE(P)) then
    /* Find a common ancestor */
    OT_GTEQ_FACE_NEIGHBOR2(FATHER(P),I,Q,L)
  else Q ← FATHER(P);
  /* Follow the reflected path to locate the neighbor */
  if not(null(Q)) and GRAY(Q) then
    begin
      Q ← SON(Q,REFLECT(I,SONTYPE(P)));
      L ← L - 1;
    end;
end;

recursive pointer node procedure OT_GTEQ_EDGE_NEIGHBOR2(P,I,Q,L);
/* Return in Q the edge-neighbor of node P, of size greater than or equal to P, in direction I. L denotes the level of the tree
   at which node P is initially found, and the level of the tree at which node Q is ultimately found. If such a node does not
   exist, then return NIL. */
```

```
begin
  value pointer node P;
  value edge I;
  reference pointer node Q;
  reference integer L;
  L ← L + 1;
  /* Find a common ancestor */
  if null(FATHER(P)) then Q ← NIL
  else if ADJ(I,SONTYPE(P)) then
    OT_GTEQ_EDGE_NEIGHBOR2(FATHER(P),I,Q,L)
  else if COMMON_FACE(I,SONTYPE(P)) ≠ Ω then
    OT_GTEQ_FACE_NEIGHBOR2(FATHER(P),COMMON_FACE(I,SONTYPE(P)),Q,L)
  else Q ← FATHER(P);
  /* Follow opposite path to locate the neighbor */
  if not(null(Q)) and GRAY(Q) then
    begin
      Q ← SON(Q,REFLECT(I,SONTYPE(P)));
      L ← L − 1;
    end;
end;

recursive pointer node procedure OT_GTEQ_VERTEX_NEIGHBOR2(P,I,Q,L);
/* Return in Q the vertex-neighbor of node P, of size greater than or equal to P, in direction I. L denotes the level of the tree
   at which node P is initially found, and the level of the tree at which node Q is ultimately found. If such a node does not
   exist, then return NIL. */
begin
  value pointer node P;
  value vertex I;
  reference pointer node Q;
  referencer integer L;
  L ← L + 1;
  /* Find a common ancestor */
  if null(FATHER(P)) then Q ← NIL
  else if ADJ(I,SONTYPE(P)) then
    OT_GTEQ_VERTEX_NEIGHBOR2(FATHER(P),I,Q,L)
  else if COMMON_EDGE(I,SONTYPE(P)) ≠ Ω then
    OT_GTEQ_EDGE_NEIGHBOR2(FATHER(P),COMMON_EDGE(I,SONTYPE(P)),Q,L)
  else if COMMON_FACE(I,SONTYPE(P)) ≠ Ω then
    OT_GTEQ_FACE_NEIGHBOR2(FATHER(P),COMMON_FACE(I,SONTYPE(P)),Q,L)
  else Q ← FATHER(P);
  /* Follow opposite path to locate the neighbor */
  return(if not(null(Q)) and GRAY(Q) then
    begin
      Q ← SON(Q,REFLECT(I,SONTYPE(P)));
      L ← L − 1;
    end;
end;
```

5. EXAMPLE

It is difficult to give an example of the algorithm in three dimensions. Thus, instead, we show below how ray R is traced through the two-dimensional scene given in Fig. 3. The algorithm, as encoded by procedure RAY_TRACER and the associated procedures, is also valid for two-dimensional scenes. The only necessary modifications are minor and are described briefly below:

1. Replace loops and data structures that cycle through 'X', 'Y', and 'Z' by just 'X' and 'Y'. Thus, FIND_3D_BLOCK is replaced by FIND_2D_BLOCK.
2. Remove variables CZX and CZY, as well as all tests involving them. This means that the conclusion of the test (*i.e.*, the action, or actions, to be taken had the test's evaluation yielded a value of true) is also removed.
3. Remove all tests involving 'Z' and the associated actions to be taken had the test's evaluation yielded a value of true.
4. Let the directions W, E, S, N correspond to L, R, D, U, respectively, and simplify Tables 6, 7, and 9. Table 8 is no longer necessary.

Continuing with our example, the scene is represented as a PM_1 quadtree in a $2^5 \times 2^5$ space with an origin at the lower left corner. The viewpoint is assumed to be at the point (−8, 23). Ray R is assumed to pass through the point (12, 16). Therefore, R is defined parametrically by

$$x = 20 \cdot t - 8,$$

$$y = -7 \cdot t + 23.$$

R first enters the scene at the point defined by $t = 2/5$, *i.e.*, (0, 101/5). This is obtained by taking the maximum of $t_x^{in} = 2/5$ computed at $x = 0$ and $t_y^{in} = -9/7$ computed at $y = 32$. The point (0, 101/5) is contained in cell 1. Cell 1 is exited at the point defined by $t = 4/5$, *i.e.*, (8, 87/5) in the easterly direction, and is obtained by taking the minimum of $t_x^{out} = 4/5$ computed

Table 6. F=FACE_DIR(A,SIGN_M[A]).

A (normal axis)	SIGN_M[A]	F (direction)
X	1	L
X	0	R
Y	1	D
Y	0	U
Z	1	B
Z	0	F

at $x = 8$, and $t_y^{out} = 1$ computed at $y = 16$. This process is repeated for the rest of the cells intersected by the ray and its result is shown in Table 10.

Values of t are tabulated as ordered pairs where 'num' and 'den' correspond to t's numerator and denominator, respectively. Notice that $t_x^{out} = t_y^{out}$ for cell 2, which means that CYX = 0 and MIN_AXIS is set to 'X' in procedure RAY_TRACER. Procedure NEXT_CELL_DIRECTION indicates that the direction of the next cell is to be found in EDGE_DIR('XY', 0, 1), *i.e.*, 'RD', which is the same as 'SE'. The 'SE' neighbor of cell 2 is cell 3 and is located by use of the two-dimensional analog of procedure OT_GTEQ_NEIGHBOR. Since cell 3 is larger than cell 2, there is no need to make use of procedure FIND_2D_BLOCK to locate it.

6. CONCLUDING REMARKS

We have given an algorithm for tracing a ray in a scene represented by an octree by using neighbor finding. As with any application in computer graphics numerical precision is an important issue. We have skirted this issue, in part, by using rational arithmetic. Such an approach is adequate as long as secondary rays do not result in the creation of more secondary rays (*e.g.*, a ray is reflected off several surfaces). The problem is that the number of bits that are required to maintain the same amount of precision grows geometrically.

Our algorithm is quite long because a correct implementation requires the consideration of many subtle points. In fact, Wyvill and Kunii[31] give a much shorter algorithm. Their algorithm has a similar structure to ours although it uses point location rather than neighbor finding. Also, it adheres to different conventions. In the following, we briefly point out how their solution could go awry.

First, let us examine more closely our convention that for a ray to pass through a cell (as well as intersect an object), it must enter and exit the cell (object) at two distinct points. As an example of what could go wrong if we don't adhere to this convention, consider Fig. 3. Assume the existence of a ray that passes from cell 5 to cell 11 through the SW corner of cell 5, and suppose that we say that the ray passes through cell 4 before reaching cell 11. Let (a_x, a_y) denote the point at which cells 5 and 11 touch. Cell 4 is the next cell intersected by the ray if we apply procedure FIND_2D_BLOCK to the point $(a_x - 1, a_y)$. Now, suppose cell 4 is completely occupied by an object such that the object's southern and eastern boundaries coincide with the southern and eastern boundaries of the cell. This means that a false object intersection will be reported. On the other hand, cell 12 is the next cell intersected by the ray if we apply FIND_2D_BLOCK to the point $(a_x, a_y - 1)$ in which case no false object intersection is reported.

Wyvill and Kunii's alternative algorithm identifies the next cell by calculating the coordinates of a point (*i.e.*, POINT) that is purportedly guaranteed to be in that cell and then locates it by use of a process similar to that given by FIND_3D_BLOCK. It does not compute a direction as done in RAY_TRACER. This method of calculating POINT is similar to the method used in RAY_TRACER with the following minor difference. It subtracts SIGN_M[I] from the numerator of the t value corresponding to the Ith coordinate of the exit point while we, instead, add the term SIGN_M[I]*CHANGE(DIRECT(C),I) in the computation of POINT[I]. Their algorithm is given below:

```
begin
  rational array T[{'X','Y','Z'}];
  axis I,K;
  for I in {'X','Y','Z'} do
    begin
      NUM(T[I]) ← CORNER[I]-SIGN_M[I]*W-B[I]-SIGN_M[I];
      DEN(T[I]) ← M[I];
    end;
  K ← 'X';
  for I in {'Y','Z'} do
    begin
      if abs(NUM(T[K])*DEN(T[I]))>abs (NUM(T[I])*DEN(T[K])) then K←I;
    end;
  for I in {'X','Y','Z'} do
    POINT[I] ← M[I]*NUM(T[K])/DEN(T[K])+B[I];
end;
```

This algorithm works for ray R in Fig. 3. However, if we modify Fig. 3 so that cell 3 is subdivided in the same way as the SE quadrant of the entire quadtree, then this algorithm can yield an erroneous result. This can be seen by tracing ray R through the modified figure. The problem is that procedure RAY_TRACER goes to cell 3 after cell 2, whereas the alternative al-

Table 7. E=EDGE_DIR(PIJ,SIGN_M[I],SIGN_M[J]).

PIJ (normal plane)	SIGN_M[I]	SIGN_M[J]	E (direction)
XY	1	1	LD
XY	1	0	LU
XY	0	1	RD
XY	0	0	RU
XZ	1	1	LB
XZ	1	0	LF
XZ	0	1	RB
XZ	0	0	RF
YZ	1	1	DB
YZ	1	0	DF
YZ	0	1	UB
YZ	0	0	UF

Table 8. V=VERTEX_DIR(SIGN_M[X],SIGN_M[Y],SIGN_M[Z]).

SIGN_M[X]	SIGN_M[Y]	SIGN_M[Z]	V (direction)
1	1	1	LDB
1	1	0	LDF
1	0	1	LUB
1	0	0	LUF
0	1	1	RDB
0	1	0	RDF
0	0	1	RUB
0	0	0	RUF

gorithm calculates POINT = (12, 16), which FIND_2D_BLOCK determines to be in cell 10. From cell 10, the alternative algorithm computes POINT = (14, 15) instead of the correct value of POINT = (12, 15). Although in the case of the original Fig. 3, FIND_2D_BLOCK determines both of these points to be in cell 3, this is not the case in the modified figure. The result is that a transition is made to the wrong cell.

One way to fix the alternative algorithm is to remove the subtraction of SIGN_M[I] from the numerator of the t value corresponding to the Ith coordinate of the exit point. Instead, SIGN_M[J] is subtracted in the computation of POINT[J] where J is the coordinate corresponding to the minimum value of t. Wyvill and Kunii claim that it is not necessary to compute the exact direction of motion. In particular, when a ray reaches more than one boundary simultaneously, they arbitrarily pick one of the boundaries, and move in a direction perpendicular to it.

Unfortunately, the above fix will not always yield the correct result. For example, in two dimensions, a motion in the SW direction is decomposed into two motions—one each in the S and W directions. This can lead to an error as described earlier in this section when we discussed the ramifications of the convention that for a ray to pass through a cell (as well as intersect an object), it must enter and exit the cell (object) at two distinct points. Thus, the only way to ensure the correctness of the alternative algorithm is to make it

Table 9. CHANGE(I,A).

I (direction)	A (axis)		
	X	Y	Z
L	-1	0	0
R	1	0	0
D	0	-1	0
U	0	1	0
B	0	0	-1
F	0	0	1
LD	-1	-1	0
LU	-1	1	0
LB	-1	0	-1
LF	-1	0	1
RD	1	-1	0
RU	1	1	0
RB	1	0	-1
RF	1	0	1
DB	0	-1	-1
DF	0	-1	1
UB	0	1	-1
UF	0	1	1
LDB	-1	-1	-1
LDF	-1	-1	1
LUB	-1	1	-1
LUF	-1	1	1
RDB	1	-1	-1
RDF	1	-1	1
RUB	1	1	-1
RUF	1	1	1

Table 10. Result of tracing ray R through Fig. 3.

cell	size	t_x^{out}		t_y^{out}		t^{out}		x_{out}	y_{out}	direction of next cell	neighbor type
		num	den	num	den	num	den				
1	8	16	20	−7	−7	16	20	8	87/5	R	E
2	4	20	20	−7	−7	20	20	12	16	RD	SE
3	16	24	20	−23	−7	24	20	16	73/5	R	E
4	8	32	20	−15	−7	32	20	24	59/5	R	E
5	4	36	20	−15	−7	36	20	28	52/5	R	E
6	4	40	20	−15	−7	40	20	32	9	R	E

identical to procedure RAY_TRACER (*i.e.*, to compute the exact direction of the neighboring cell relative to the current cell).

Acknowledgements—I have benefitted greatly from discussions with Robert E. Webber. The support of the National Science Foundation under Grant IRI-88-02457 is gratefully acknowledged.

REFERENCES

1. D. Ayala, P. Brunet, R. Juan and I. Navazo, Object representation by means of nonminimal division quadtrees and octrees. *ACM Trans. on Graphics* **4** (1), 41–59 (January 1985).
2. I. Carlbom, I. Chakravarty and D. Vanderschel, A hierarchical data structure for representing the spatial decomposition of 3-D objects. *IEEE Comp. Graphics and Appl.* **5** (4), 24–31 (April 1985).
3. M. Cyrus and J. Beck, Generalized two- and three-dimensional clipping. *Comp. & Graphics* **3** (1), 23–28 (1978).
4. K. Fujimura and T. L. Kunii, A hierarchical space indexing method. *Proceedings of Comp. Graphics* '85, Tokyo, T1-4, 1–14 (1985).
5. A. Fujimoto, T. Tanaka and K. Iwata, ARTS: Accelerated ray-tracing system. *IEEE Comp. Graphics and Appl.* **6** (4), 16–26 (April 1986).
6. I. Gargantini, Linear octrees for fast processing of three-dimensional objects. *Comp. Graphics and Image Processing* **20** (4), 365–374 (December 1982).
7. A. S. Glassner, Space subdivision for fast ray tracing. *IEEE Comp. Graphics and Appl.* **4** (10), 15–22 (October 1984).
8. G. M. Hunter, Efficient computation and data structures for graphics. Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ (1978).
9. G. M. Hunter, Geometrees for interactive visualization of geology: An evaluation, System Science Department, Schlumberger-Doll Research, Ridgefield, CT (1981).
10. C. L. Jackins and S. L. Tanimoto, Oct-trees and their use in representing three-dimensional objects. *Comp. Graphics and Image Processing* **14** (3), 249–270 (November 1980).
11. F. W. Jansen, Data structures for ray tracing. In F. J. Peters, L. R. A. Kessner and M. L. P. van Lierop (Eds.), *Data Structures for Raster Graphics*, 57–73, Springer-Verlag, Berlin (1986).
12. M. R. Kaplan, Space-tracing: A constant time ray-tracer, SIGGRAPH'85 Tutorial on the Uses of Spatial Coherence in Ray-Tracing, San Francisco, ACM (July 1985).
13. M. Kaplan, The use of spatial coherence in ray tracing. In D. F. Rogers and R. A. Earnshaw (Eds.), *Techniques for Computer Graphics*, 173–193, Springer-Verlag, New York (1987).
14. A. Klinger, Patterns and search statistics. In J. S. Rustagi (Ed.), *Optimizing Methods in Statistics*, 303–337, Academic Press, New York (1971).
15. D. Meagher, Geometric modeling using octree encoding. *Comp. Graphics and Image Processing* **19** (2), 129–147. (June 1982).
16. I. Navazo, Contribució a les tècniques de modelat geomètric d'objectes poliédrics usant la codificació amb arbres octals. Ph.D. Dissertation, Escola Tecnica Superior d'Enginyers Industrials, Department de Metodes Informatics, Universitat Politechnica de Barcelona, Barcelona, Spain (January 1986).
17. I. Navazo, D. Ayala and P. Brunet, A geometric modeller based on the exact octree representation of polyhedra. *Comp. Graphics Forum* **5** (2), 91–104 (June 1986).
18. K. M. Quinlan and J. R. Woodwark, A spatially-segmented solids database—justification and design. *Proceedings of CAD'82 Conference*, Butterworth, Guildford, Great Britain, 126–132 (1982).
19. D. F. Rogers, *Procedural Elements for Computer Graphics*, McGraw-Hill, New York (1985).
20. H. Samet, Neighbor finding techniques for images represented by quadtrees. *Comp. Graphics and Image Processing* **18** (1), 37–57 (January 1982).
21. H. Samet, The quadtree and related hierarchical data structures. *ACM Comp. Surveys* **16** (2), 187–260 (June 1984).
22. H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA (1990).
23. H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA (1990).
24. H. Samet, Neighbor finding in images represented by octrees. *Comp. Vision, Graphics, and Image Processing* **46** (3), 367–386 (June 1989).
25. H. Samet and R. E. Webber, Storing a collection of polygons using quadtrees. *ACM Trans. on Graphics* **4** (3), 182–222 (July 1985). (Also *Proceedings of Computer Vision and Pattern Recognition* 83, Washington, DC, 127–132 June 1983; and University of Maryland Computer Science TR-1372).
26. H. Samet and R. E. Webber, Hierarchical data structures and algorithms for computer graphics. Part I. Fundamentals. *IEEE Comp. Graphics and Appl.* **8** (3), 48–68 (May 1988).
27. H. Samet and R. E. Webber, Hierarchical data structures and algorithms for computer graphics. Part II. Applications. *IEEE Comp. Graphics and Appl.* **8** (4), 59–75 (July 1988).
28. M. Tamminen, The EXCELL method for efficient geometric access to data, *Acta Polytechnica Scandinavica*. Mathematics and Computer Science Series No. 34, Helsinki, Finland (1981).
29. D. J. Vanderschel, Divided leaf octal trees. Research Note, Schlumberger-Doll Research, Ridgefield, CT (March 1984).
30. T. Whitted, An improved illumination model for shaded display. *Comm. of the ACM* **23** (6), 343–349 (June 1980).
31. G. Wyvill and T. L. Kunii, A functional model for constructive solid geometry. *Visual Comp.* **1** (1), 3–14 (July 1985).