COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING 46, 367–386 (1989)

Neighbor Finding in Images Represented by Octrees*

HANAN SAMET

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

Received January 11, 1988; accepted November 21, 1988

Algorithms are presented for moving between adjacent blocks in an octree representation of an image. Motion is possible in the direction of a face, edge, and a vertex, and between blocks of arbitrary size. The algorithms are based on a generalization and simplification of techniques developed earlier for two dimensions (i.e., in quadtrees). They are also applicable to quadtrees. The difference lies in the graph-theoretical classification of adjacencies—i.e., in terms of vertices, edges, and faces. Algorithms are given for octrees that are implemented with pointers and with pointerless representations such as the linear octree. © 1989 Academic Press, Inc.

1. INTRODUCTION

Image representation is an important issue in computer vision, computer graphics, and image processing. There are many different representations in use. Currently there is much interest in the use of hierarchical data structures such as the quadtree and octree [19, 20, 21, 25, 26]. Quadtrees are used in 2-dimensional applications and octrees in 3-dimensional applications. Their development has been motivated to a large extent by a desire to save storage by aggregating data having identical or similar values. However, this is not always the case, and, in fact, the savings in execution time that arise from this aggregation are often of equal or greater importance.

Many of the traditional image processing operations can be performed using quadtrees and octrees. For example, computing perimeters [17], labeling connected components [16], finding the genus of an image [2], and computing set properties [7, 27]. Operations in three dimensions are also possible [9, 1, 28, 29]. The operations are frequently implemented as tree traversals. The difference between them is in the nature of the computation that is performed at the node. Often, these computations involve the examination of nodes whose corresponding blocks are spatially "adjacent" to the block corresponding to the node being processed. We shall speak of these adjacent nodes as "neighbors." However, we must be careful to note that adjacency in space does not imply any simple relationship among the nodes in the tree.

In this paper we show how to move between adjacent blocks in an octree. Motion is possible in the direction of a face, edge, and a vertex, and between blocks of arbitrary size. Thus unlike others we do not restrict ourselves to motion in the direction of a face between voxels of equal size (e.g., [3, 13]) which is quite easy as it is analogous to motion in the direction of an edge in two dimensions. Our method is a generalization and a simplification of the techniques we developed earlier for two dimensions (i.e., in quadtrees) [18, 23]. The difference is in the graph-theoretical

*The support of the National Science Foundation under Grant IRZ-88-02457 is gratefully acknowledged.

0734-189X/89 $3.00

classification of adjacencies—i.e., in terms of vertices, edges, and faces. It is also applicable to quadtrees, and it can be easily extended to deal with images of higher dimension. We show how it can be used for octrees that are implemented using both a pointer and a pointer-less representation such as the linear octree [3].

The rest of this paper is organized as follows. Section 2 contains some definitions and a description of our notation. Section 3 presents the algorithms for computing the neighbors. The algorithms are given for an octree representation that uses pointers. Section 4 analyzes their execution time. Section 5 adapts them to a pointer-less quadtree representation. Section 6 concludes with a brief discussion of the use of these algorithms with quadtrees and their extension to higher dimensional data.

The motivation for this paper was a desire to implement a ray tracing algorithm that made use of an octree representation of a scene. Ray tracing is an approximate simulation of how the light that is propagated through a scene lands on the image plane [15]. This task is useful in computer graphics. However, it is also the basis of performing line of sight (in an arbitrary direction) and field of view computations in determining visibility in applications in computer vision and robotics. Although octrees are frequently used in such applications [5, 10, 30], none of these methods uses neighbor finding to trace the ray through the scene. Instead, more complex methods are used that involve postulating a point lying outside of a given node, computing its address, and then searching through the tree for it. As we will see, the method we describe is simpler.

2. DEFINITIONS AND NOTATION

The *region octree* [6, 8, 14] is an extension of the quadtree data structure [12] to represent 3-dimensional data. We start with a $2^n \times 2^n \times 2^n$ object array of unit cubes (termed *voxels* or *obels*). The octree is based on the successive subdivision of an object array into octants. If the array does not consist entirely of 1's or entirely of 0's, then it is subdivided into octants, suboctants, etc., until cubes (possibly single voxels) are obtained that consist of 1's or of 0's; 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 non-leaf nodes are said to be GRAY. Figure 1a is an example of a simple 3-dimensional object, in the form of a staircase, whose octree block decomposition is given in Fig. 1b, and whose tree representation is given in Fig. 1c.

In order to understand the presentation of the algorithms in this section, we first give some definitions and explain our notation. Figure 2 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. Figure 3 illustrates the labelings corresponding to the partitions.

The labelings in Fig. 3 are also used to identify the faces, edges, and vertices of the cube as shown in Fig. 4. The faces are L (left), R (right), D (down), U (up), B

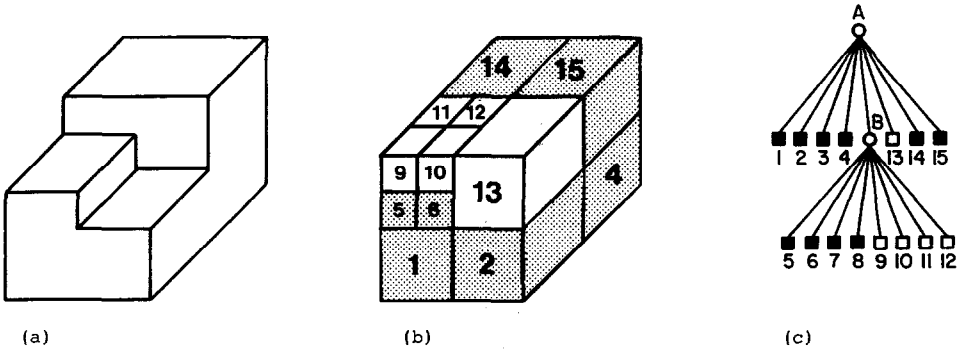

FIG. 1. (a) Example 3-dimensional object; (b) its octree block decomposition; and (c) its tree representation.

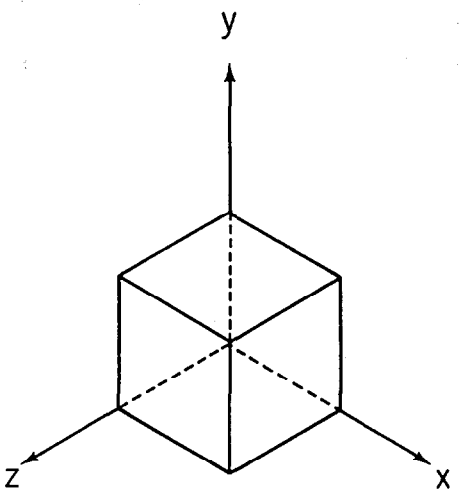

FIG. 2. *Three-dimensional coordinate system.*

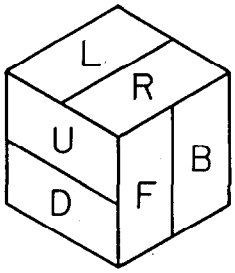

FIG. 3. Three orthogonal partitions of a cube.

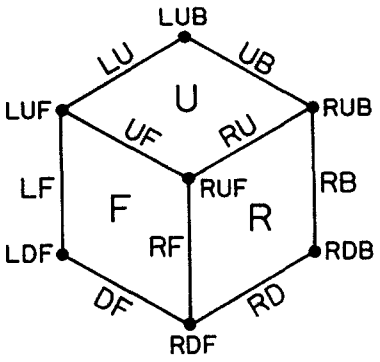

FIG. 4. Labeling of faces, edges, and vertices based on the partitioning defined in Fig. 3.

(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*, *LB*, and *DB* are not visible. Similarly, the octants are labeled by using a concatenation of these labels as shown in Fig. 5 (octant *LDB* is not visible). Figure 6 is a numerical labeling for the octants (octant 0 is not visible).

The concept of a neighbor in an octree is defined analogously to that in a quadtree. We say that node Q is a *neighbor* of node R in direction I if Q

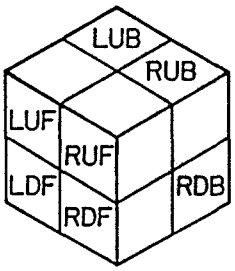

FIG. 5. Labeling of octants based on the partitioning defined in Fig. 3 (octant LDB is not visible).

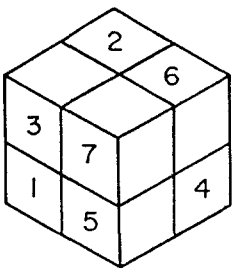

FIG. 6. Numeric labeling of octants based on the partitioning defined in Fig. 3 (octant 0 is not visible).

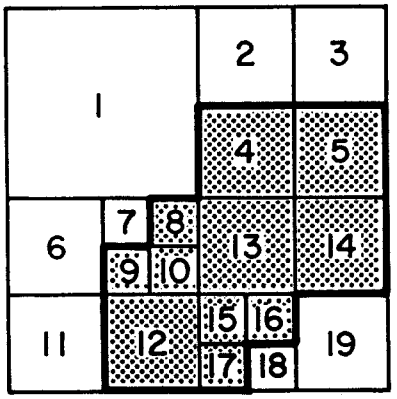

FIG. 7. Example of a quadtree block decomposition.

corresponds to the smallest block (it may correspond to a non-leaf 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. For example, in two dimensions, a node can have a minimum of five neighbors (e.g., the node corresponding to block 7 in Fig. 7), and a maximum of eight neighbors (e.g., the node corresponding to block 13 in Fig. 7, where the southern and western neighbors are non-leaf nodes). The neighbors of a node R can be of any color but all of them cannot have the same color as R as then a merge would have occurred.

However, whereas in two dimensions we have eight possible directions, in three dimensions, we have 26 possible directions. In particular, in two dimensions, two nodes can be adjacent, and hence neighbors, along an edge (four possibilities) or along a vertex (four possibilities). In contrast, in three dimensions, two nodes can be adjacent, and hence neighbors, along a face (six possibilities), along an edge (12 possibilities), or along a vertex (eight possibilities). Such neighbors are termed face-neighbors, edge-neighbors, and vertex-neighbors, respectively. These relations are shown in Figs. 8a, 8b, and 8c, respectively.

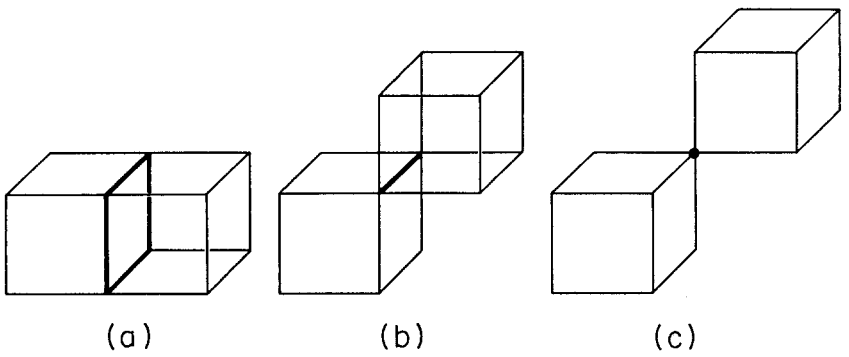

FIG. 8. Example of (a) a face neighbor, (b) an edge neighbor, and (c) a vertex neighbor.

TABLE 1
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

We now describe an octree implementation that uses pointers. In Section 5 we describe a pointerless octree implementation. Assume that each octree node is stored as a record of type *node* containing ten 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) 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 tenth field, NODETYPE, describes 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. It is exploited in a number of algorithms in order to perform the basic operations.

The predicate ADJ, and the functions REFLECT, COMMON_FACE, and COMMON_EDGE aid in the expression of operations involving a block's octants and its faces, edges, and vertices. Tables 1–4 contain the definitions of the ADJ, REFLECT, COMMON_FACE, and COMMON_EDGE relationships, respectively. Ω denotes an undefined value.

TABLE 2
REFLECT(I, O)

	O (octant)							
I (direction)	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

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—e.g., ADJ('L', 'LDB') is true as are ADJ('LD', 'LDB') and ADJ('LDB', 'LDB'). This relation can also be described as being true if O is of type I, or equivalently that I's label is a subset of O's label.

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—e.g., REFLECT('L', 'RDB') = 'LDB', REFLECT('LD', 'RDB') = 'LUB', and REFLECT('RDB', 'RDB') = 'LUF'.

COMMON_FACE(I, O) yields the type on 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—e.g., COMMON_FACE('LD', 'LUF') = 'L' and COMMON_FACE('LDB', 'LUF') = 'L'.

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)—e.g., COMMON_EDGE('LDB', 'LUB') = 'LB'.

For an octree corresponding to a $2^n \times 2^n \times 2^n$ image array, we say that the root is at level n, and that a node at level i is at a distance of $n - i$ from the root of the tree. In other words, for a node at level i, we must ascend $(n - i)$ FATHER links to reach the root of the tree. Note that the farthest node from the root of the tree is at

TABLE 3
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	Ω
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

level ≥ 0. A node at level 0 corresponds to a single voxel in the image, while a node is of size 2^s if it is found at level s in the tree.

3. ALGORITHMS

First, we show how to locate neighbors of equal size. This is relatively straightforward for face-neighbors. Assume that we are trying to find the equal-sized face-neighbor of node P in direction I. The basic idea is to ascend the octree until a common ancestor is located, and then descend back down the octree in search of the neighboring node. It is obvious that we can always ascend as far as the root of the

TABLE 4
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

octree and then start our descent. However, our goal is to find the nearest common ancestor, as this minimizes the number of nodes that must be visited. These two steps are described below. They are implemented by procedure OT_EQ_FACE_NEIGHBOR. This procedure ignores the situation in which the neighbor may not exist (e.g., when the node is on the border of the image).

(1) Locate the nearest common ancestor. This is the first ancestor node reached by a son of type O such that ADJ(I, O) is false. In other words, I's label is not a subset of O's label.

(2) Retrace the path that was taken to locate the nearest common ancestor by using the function REFLECT to make mirror image moves about the face shared by the neighboring nodes.

```
recursive pointer node procedure OT_EQ_FACE_NEIGHBOR(P, I);
/* Locate an equal-sized face-neighbor of node P in direction I. */
begin
  value pointer node P;
  value face I;
  return(SON(if ADJ(I,SONTYPE(P)) then
                OT_EQ_FACE_NEIGHBOR(FATHER(P),I)
              else FATHER(P),
              REFLECT(I,SONTYPE(P))));
end;
```

Locating an edge-neighbor is more complex. Assume that we are trying to find the equal-sized edge-neighbor of node P in direction I. Our initial aim is to locate the nearest common ancestor of P and its neighbor. We need to ascend the tree to do so. We must also account for the situation in which the ancestors of P and its neighbor are adjacent along a face. Let N denote the node that is currently being examined in the ascent. There are three cases described below. They are implemented by procedure OT_EQ_EDGE_NEIGHBOR. This procedure ignores the situation in which the neighbor may not exist (e.g., when the node is on the border of the image).

(1) As long as N is a son of type O such that ADJ(I, O) is true, we continue to ascend the tree. In other words, I's label is a subset of O's label.

(2) If the father of N and the ancestor of the desired neighbor, say A, are adjacent along a face, then calculate A by use of procedure OT_EQ_FACE_NEIGHBOR. This situation and the exact direction of A are determined by the function COMMON_FACE applied to I and the son type of N. Once A has been obtained, the desired neighbor is located by applying the retracing step outlined in step 3.

(3) Otherwise, N is a son of a type, say O, such that neither of the labels of the faces comprising edge I is a subset of O's label. Its father, say T, is the nearest common ancestor. The desired neighbor is obtained by simply retracing the path used to locate T, except that we now make directly opposite moves about the edge shared by the neighboring nodes. This process is facilitated by use of the function REFLECT.

```
recursive pointer node procedure OT_EQ_EDGE_NEIGHBOR(P, I);
/* Locate an equal-sized edge-neighbor of node P in direction I. */
```

```
begin
  value pointer node P;
  value edge I;
  return(SON(if ADJ(I, SONTYPE(P)) then
                OT_EQ_EDGE_NEIGHBOR(FATHER(P), I)
              else if COMMON_FACE(I,SONTYPE(P)) ≠ Ω then
                OT_EQ_FACE_NEIGHBOR(FATHER(P),
                                    COMMON_FACE(I,SONTYPE(P)))
              else FATHER(P),
              REFLECT(I,SONTYPE(P)))):
end;
```

Locating a vertex-neighbor is very similar to locating an edge-neighbor. Assume that we are trying to find the equal-sized vertex-neighbor of node P in direction I. Again, our initial aim is to locate the nearest common ancestor of P and its neighbor. We need to ascend the tree to do so. We must also account for the situation that the ancestors of P and its neighbor are adjacent along an edge. Let N denote the node that is currently being examined in the ascent. There are four cases described below. They are implemented by procedure OT_EQ_VERTEX _NEIGHBOR. This procedure ignores the situation that the neighbor may not exist (e.g., when the node is on the border of the image).

(1) As long as N is a son of type O such that ADJ(I, O) is true, we continue to ascend the tree. In other words, I's label is a subset of O's label.

(2) If the father of N and the ancestor of the desired neighbor, say A, are adjacent along an edge, then calculate A by use of procedure OT_EQ_ EDGE_NEIGHBOR. This situation and the exact direction of A are determined by the function COMMON_EDGE applied to I and the son type of N. Once A has been obtained, the desired neighbor is located by applying the retracing step outlined in step 4.

(3) If the father of N and the ancestor of the desired neighbor, say A, are adjacent along a face, then calculate A by use of procedure OT_EQ_ FACE_NEIGHBOR. This situation and the exact direction of A are determined by the function COMMON_FACE applied to I and the son type of N. Once A has been obtained, the desired neighbor is located by applying the retracing step outlined in step 4.

(4) Otherwise, N is a son of a type, say O, such that none of the labels of the faces comprising vertex I is a subset of O's label. Its father, say T, is the nearest common ancestor. The desired neighbor is obtained by simply retracing the path used to locate T except that we now make directly opposite moves about the vertex shared by the neighboring nodes. This process is facilitated by use of the function REFLECT.

```
recursive pointer node procedure OT_EQ_VERTEX_NEIGHBOR(P,I);
/* Locate an equal-sized vertex-neighbor of node P in direction I. */
begin
  value pointer node P;
  value vertex I;
  return(SON(if ADJ(I,SONTYPE(P)) then
                OT_EQ_VERTEX_NEIGHBOR(FATHER(P),I)
              else if COMMON_EDGE(I,SONTYPE(P)) ≠ Ω then
                OT_EQ_EDGE_NEIGHBOR(FATHER(P),
                                    COMMON_EDGE(I,SONTYPE(P)))
```

```
            else if COMMON_FACE(I,SONTYPE(P)) ≠ Ω then
              OT_EQ_FACE_NEIGHBOR(FATHER(P),
                                  COMMON_FACE(I,SONTYPE(P)))
            else FATHER(P),
            REFLECT(I,SONTYPE(P))));
end;
```

In general, neighbors need not correspond to blocks of the same size. If the neighbor is larger, then only part of the path from the nearest common ancestor is retraced. Otherwise the neighbor corresponds to a block of equal size and a pointer to a BLACK, WHITE, or GRAY node, as is appropriate, of equal size is returned. If there is no neighbor (i.e., the node whose neighbor is being sought is adjacent to the border of the image in the specified direction), then NIL is returned. This process is encoded below by procedures OT_GTEQ_FACE_NEIGHBOR, OT_GTEQ_EDGE_NEIGHBOR, and OT_GTEQ_VERTEX_NEIGHBOR. They replace OT_EQ_FACE_NEIGHBOR, OT_EQ_EDGE_NEIGHBOR, and OT_EQ_VERTEX_NEIGHBOR, respectively.

```
recursive pointer node procedure OT_GTEQ_FACE_NEIGHBOR(P,I);
/* Locate a face-neighbor of node P, of size greater than or equal to P, in direction I. If such
   a node does not exist, then return NIL. */
begin
  value pointer node P;
  value face I;
  pointer node Q;
  Q ← if not null(FATHER(P)) and ADJ(I,SONTYPE(P)) then
        /* Find a common ancestor */
        OT_GTEQ_FACE_NEIGHBOR(FATHER(P),I)
      else FATHER(P);
  /* Follow the reflected path to locate the neighbor */
  return(if not null(Q) and GRAY(Q) then SON(Q,REFLECT(I,SONTYPE(P)))
         else Q);
end;

recursive pointer node procedure OT_GTEQ_EDGE_NEIGHBOR(P,I);
/* Locate an edge-neighbor of node P, of size greater than or equal to P, in direction I. If
   such a node does not exist, then return NIL. */
begin
  value pointer node P;
  value edge I;
  pointer node Q;
  /* Find a common ancestor */
  Q ← if null(FATHER(P)) then NIL
      else if ADJ(I,SONTYPE(P)) then
        OT_GTEQ_EDGE_NEIGHBOR(FATHER(P),I)
      else if COMMON_FACE(I,SONTYPE(P)) ≠ Ω then
        OT_GTEQ_FACE_NEIGHBOR(FATHER(P),
                              COMMON_FACE(I,SONTYPE(P)))
      else FATHER(P);
  /* Follow opposite path to locate neighbor */
  return(if not null(Q) and GRAY(Q) then SON(Q,REFLECT(I,SONTYPE(P)))
         else Q);
end;

recursive pointer node procedure OT_GTEQ_VERTEX_NEIGHBOR(P,I);
/* Locate a vertex-neighbor of node P, of size greater than or equal to P, in direction I. If
   such a node does not exist, then return NIL. */
```

```
begin
  value pointer node P;
  value vertex I;
  pointer node Q;
  /* Find a common ancestor */
  Q ← if null(FATHER(P)) then NIL
       else if ADJ(I,SONTYPE(P)) then
         OT_GTEQ_VERTEX_NEIGHBOR(FATHER(P),I)
       else if COMMON_EDGE(I,SONTYPE(P)) ≠ Ω then
         OT_GTEQ_EDGE_NEIGHBOR(FATHER(P),
                                COMMON_EDGE(I,SONTYPE(P)))
       else if COMMON_FACE(I,SONTYPE(P)) ≠ Ω then
         OT_GTEQ_FACE_NEIGHBOR(FATHER(P),
                                COMMON_FACE(I,SONTYPE(P)))
       else FATHER(P);
  /* Follow opposite path to locate the neighbor */
  return(if not null(Q) and GRAY(Q) then
           SON(Q,REFLECT(I,SONTYPE(P)))
         else Q);
end;
```

4. ANALYSIS

The analysis of the algorithms that we have presented depends on the assumptions about the underlying random image model. The most natural way to analyze the execution time of these functions is in terms of the number of nodes that must be visited in locating the desired neighbor [18, 23]. The analysis of each function is decomposed into two stages. They correspond to the process of locating the nearest common ancestor, and then locating the desired neighbor. The worst-case analysis is $O(n)$. However, it is rare. We use an average case-analysis that makes use of a random image model in which each leaf node is assumed to be equally likely to appear at any position and level in the octree. Using such a model with an octree means that there are $1, 8, 64, 512, \ldots, 8^i$ leaf nodes at levels $n, n-1$, $n-2, n-3, \ldots, n-i$, respectively, or equivalently that $\frac{7}{8} \cdot (\frac{1}{8})^i$ of the nodes are at level i. Of course, this is not a realizable situation, although in practice it does model the image well [18, 23].

Observe that our notion of a random image differs from the conventional one which implies that every voxel has an equal probability of being BLACK or WHITE. Such an assumption leads to a very low probability of aggregation (i.e., nodes corresponding to blocks of size greater than one voxel). Clearly, for such an image the octree is the wrong representation (e.g., a checkerboard-like solid). The problem with the conventional random image model is that it assumes independence which is clearly not the case (i.e., a voxel's value is typically related to that of its neighbors).

The first stage of the analysis is with respect to a node P at level i and a direction I. Depending on the nature of I, there are different positions where P might be located. If I is a face, then there are $(2^{n-i})^2 \cdot (2^{n-i} - 1)$ possible positions where P might be located. The -1 results from the fact that along one of the axial directions, one element will not have a neighbor in direction I. Of these positions, $2^{n-i} \cdot 2^{n-i} \cdot 2^0$ have their nearest common ancestor at level n, $2^{n-i} \cdot 2^{n-i} \cdot 2^1$ at level $n-1, \ldots,$ and $2^{n-i} \cdot 2^{n-i} \cdot 2^{n-i-1}$ at level $i+1$. To reach a nearest common

ancestor at level j, $j - i$ nodes must be visited. Therefore, the average is

$$\frac{\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} 2^{n-i} \cdot 2^{n-i} \cdot 2^{n-j} \cdot (j - i)}{\sum_{i=0}^{n-1} 2^{n-i} \cdot 2^{n-i} \cdot (2^{n-i} - 1)}.$$

This can be simplified to yield

$$2 - \frac{(21n - 7) \cdot 2^{2n} + 7}{18 \cdot 2^{3n} - 21 \cdot 2^{2n} + 3} \leq 2 \qquad \text{for } n \geq 1.$$

If I is an edge, then there are $2^{n-i} \cdot (2^{n-i} - 1)^2$ possible positions where P might be located. The -1 results from the fact that along two of the axial directions, one element will not have a neighbor in direction I that is in the image. Of these positions, $2^{n-i} \cdot 4^0 \cdot (2 \cdot (2^{n-i} - 1) - 1)$ have neighbors in direction I such that the nearest common ancestor is at level n, $2^{n-i} \cdot 4^1 \cdot (2 \cdot (2^{n-i-1} - 1) - 1)$ at level $n - 1, \ldots,$ and $2^{n-i} \cdot 4^{n-i-1} \cdot (2 \cdot (2^{n-i-(n-i-1)} - 1) - 1)$ at level $i + 1$. To reach a nearest common ancestor at level j, $j - i$ nodes must be visted. Therefore, the average is

$$\frac{\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} 2^{n-i} \cdot 4^{n-j} \cdot \left(2 \cdot (2^{n-i-(n-j)} - 1) - 1\right) \cdot (j - i)}{\sum_{i=0}^{n-1} 2^{n-i} \cdot (2^{n-i} - 1)^2}.$$

This can be simplified to yield

$$\frac{8}{3} - \frac{(28n - 28) \cdot 2^{2n} - (21n - 49) \cdot 2^n - 21}{12 \cdot 2^{3n} - 28 \cdot 2^{2n} + 21 \cdot 2^n - 5} \leq \frac{8}{3} \qquad \text{for } n \geq 1.$$

If I is a vertex, then there are $(2^{n-i} - 1)^3$ possible positions where P might be located. The -1 results from the fact that along each of the axial directions, one element will not have a neighbor in direction I that is in the image. Of these positions, $8^0 \cdot (3 \cdot (2^{n-i} - 1)^2 - 3 \cdot (2^{n-i} - 1) + 1)$ have neighbors in direction I such that the nearest common ancestor is at level n, $8^1 \cdot (3 \cdot (2^{n-i-1} - 1)^2 - 3 \cdot (2^{n-i-1} - 1) + 1)$ at level $n - 1, \ldots,$ and $8^{n-i-1} \cdot (3 \cdot (2^{n-i-(n-i-1)} - 1)^2 - 3 \cdot (2^{n-i-(n-i-1)} - 1) + 1)$ at level $i + 1$. To reach a nearest common ancestor at level

l, $j - i$ nodes must be visited. Therefore, the average is

$$\frac{\sum_{i=0}^{n-1} \sum_{j=i+1}^{n} 8^{n-j} \cdot \left(3 \cdot (2^{n-i-(n-j)} - 1)^2 - 3 \cdot (2^{n-i-(n-j)} - 1) + 1\right) \cdot (j - i)}{\sum_{i=0}^{n-1} (2^{n-i} - 1)^3}.$$

This can be simplified to yield

$$\frac{22}{7} - \frac{(147n - 217) \cdot 2^{2n+3} - (441n - 1239) \cdot 2^{n+2} + 147 \cdot n^2 - 441n - 3220}{21 \cdot 2^{3n+4} - 147 \cdot 2^{2n+3} + 441 \cdot 2^{n+2} - 294n - 924} \leq \frac{22}{7} \quad \text{for } n \geq 1.$$

When the neighbors are of equal size, then the analyses for the second stage are the same as for the first stage. The result is that the average number of nodes visited by OT_EQ_FACE_NEIGHBOR, OT_EQ_EDGE_NEIGHBOR, and OT_EQ_VERTEX_NEIGHBOR is bounded by 4, $\frac{16}{3}$, and $\frac{44}{7}$, respectively.

When the neighbors are not of equal size, then, depending on the position of the node P, a number of different sizes of neighbors are possible. For example, suppose we have a $2^3 \times 2^3 \times 2^3$ image and we are looking at a node of size one voxel (i.e., $1 \times 1 \times 1$). In this case, its position may be such that it can have three possible neighbors—i.e., one each of size 1×1, 2×2, and 4×4 at node distances of 3, 2, and 1, respectively, from the nearest common ancestor. We use the average contribution of these three cases as the cost of the second stage. This means that the quantity $(j - i)$ in the analyses of OT_EQ_FACE_NEIGHBOR, OT_EQ_EDGE_NEIGHBOR, and OT_EQ_VERTEX_NEIGHBOR is replaced by

$$\frac{1}{j - i} \cdot \sum_{k=i}^{j-1} (j - k).$$

Using this model, the average number of nodes visted by OT_GTEQ_FACE_NEIGHBOR, OT_GTEQ_EDGE_NEIGHBOR, and OT_GTEQ_VERTEX_NEIGHBOR is bounded by $\frac{7}{2}$, $\frac{9}{2}$, and $\frac{73}{14}$, respectively.

5. POINTERLESS OCTREE REPRESENTATIONS

Pointerless octree representations are used because they may lead to significant savings in space as there is no need to store pointers (but see [24]). They can be grouped into two categories. The first treats the image as a collection of leaf nodes while the second represents the image in the form of a traversal of the nodes of its tree. Their disadvantage is that performing neighbor finding is more complex than when using a pointer-based representation. In this section we concentrate on a variant of the first category which is known as a linear octree [4]. We show how a neighbor of greater than or equal size is computed in the direction of a face, edge, or vertex.

The second category is exemplified by the DF-expression [11] (denoting depth-first). Neighbor finding is very complicated when this representation is used since

there is no notion of random access. For example, a node is only identified by its position in the traversal of the quadtree and is represented by a code corresponding to its type (i.e., BLACK, WHITE, or GRAY). There is no explicit information on the path from the root of the octree to it. Thus in order to perform neighbor finding, it is necessary to start at the root of the tree and locate the node whose neighbor we are seeking, remember the path to it, compute the path to its neighbor, and then sequentially search the list for the neighbor starting at the root of the tree. This is a cumbersome process.

When an image is represented as a collection of the leaf nodes comprising it, there are a number of methods of representing the individual leaf nodes. In the linear octree, each leaf node (BLACK and WHITE) is represented by its locational code. Assuming an image of side length 2^n, the locational code of each leaf node of side length 2^k is n digits long where the leading $n - k$ digits contain the directional codes that locate the leaf along a path from the root of the tree. The k trailing digits contain the directional codes that locate the voxel in the block's LDB corner. The resulting number is the same as that which is obtained by interleaving the bits that comprise the values of the x, y, and z coordinates. Therefore, each leaf node in the octree is encoded by a base 8 number.

The directional codes are numeric equivalents of the different octants (i.e., 0, 1, 2, 3, 4, 5, 6, 7 for LDB, LDF, LUB, LUF, RDB, RDF, RUB, RUF, respectively). We assume that the origin is in the LDB corner of the block and thus for a block of side length 2^k, the directional codes stored in the k trailing digits are 0.

A node's locational code is implemented by a record of type *locationalcode* with three fields PATH, LEV, and COL corresponding to the path from the root to the node, the level of the node, and the color of the node, respectively. For an image of maximum side length 2^n, the path, say P, is implemented as an array of type *path* of n directional codes stored in the order $P[n-1]P[n-2]\cdots P[1]P[0]$.

Neighbor finding when the image is represented as a collection of the locational codes of its constituent nodes is very similar to the procedure used for a pointer-based octree representation. The key difference is that there is never a need to traverse links since the only operation is one of bit manipulation. This difference is of no consequence when the tree is represented in internal memory. However, when storage requirements are such that the tree is represented in external memory (e.g., using a B-tree as in [22]), then this difference is very important for it means that there is no need to traverse links between nodes that may be on different pages, a situation that could cause a page fault. In such a situation, bit manipulation will, in most cases, be considerably faster than link traversal.

Given a node A with locational code P, finding its neighbor in direction I of size greater than or equal to A is quite simple. During this process, we first construct T_1 the path component of the locational code of the equal-sized desired neighbor, say Q. The result is analogous to the calculation of the address of the node's block except that it is more complex since we are not dealing with the individual values of the x, y, and z coordinates of the address.

The algorithm is as follows. Starting with the digit position in the path corresponding to the link from A to its father (i.e., PATH(P)[LEV(P)]), reflect each directional code in the designated direction (and assign it to the corresponding entry in T) until encountering the nearest common ancestor of A and Q. The nearest

common ancestor is detected by using the function ADJ. Note that unlike the algorithm for computing neighbors in a pointer-based octree, there is no need to descend the tree since reflection occurs while ascending the tree.

Once T has been computed, we must still determine if such a node actually exists, as well as its color. Recall that we are interested in the neighbor of greater than or equal size, while T is the path to a neighboring node of equal size. Assume, without loss of generality, that the collection of locational codes, say L, is implemented as a list. Search L for the locational code whose path has the maximum value that is still less than or equal to that of T, say R—i.e., PATH(R) $\leq T$. If the level of R is greater than or equal to that of T (i.e., LEV(P)), then R's node contains the node represented by T, and R is returned as the neighbor. Otherwise, there is no leaf node in the tree that corresponds to the desired neighbor, and the neighbor is GRAY.

Given a $2^n \times 2^n \times 2^n$ image whose octree contains m leaf nodes, the neighbor computation process described above has a worst-case execution time of $O(\log_2 m + n)$. This assumes that the locational codes are stored in a list sorted by the numbers formed by the paths to the nodes.

Procedure LC_OT_GTEQ_NEIGHBOR, given below, is used to calculate the locational code of the neighbor of a node, say A with locational code P, in all directions (i.e., face, edge, and vertex) in a linear octree. It makes use of procedures LC_OT_EQ_FACE_NEIGHBOR, LC_OT_EQ_EDGE_NEIGHBOR, and LC_OT_EQ_VERTEX_NEIGHBOR to calculate the locational code of an equal-sized neighbor of A in the required direction. These procedures are quite general and can be used with different variants of locational codes. Note the use of arrays as values that can be transmitted as parameters to procedures as well as returned as the values of procedures.

LC_OT_GTEQ_NEIGHBOR uses procedure MAXLEQ (not given here) to find the locational code of a node whose path has the maximum value that is still less than or equal to that of the path to A's equal-sized neighbor. If the node is smaller than A, then the locational code of an appropriate GRAY node is returned. The function TYPE aids in the determination of the type of the neighbor's direction (i.e., vertex, edge, or face). In order to facilitate the bit manipulation operations, the second argument to the functions ADJ, REFLECT, COMMON_EDGE, and COMMON_FACE is now the numeric code of the octant. Similarly, the value of the REFLECT function is the numeric code of the octant.

```
locationalcode procedure LC_OT_GTEQ_NEIGHBOR(N,L,I,P);
/* Given a 2^N × 2^N × 2^N image represented by a linear octree in the form of a list L, of the
   locational codes of its nodes, return the locational code of the neighbor in direction I of
   a node with locational code P. If no neighbor exists, then NIL is returned. As-
   sume N > 0. */
begin
 value integer N;
 value pointer list L;
 value direction I;
 value pointer locationalcode P;
 path array T[0 : N − 1];
 pointer locationalcode B,Q;
 if LEV(P) ≥ N then return(NIL);/* No neighbor exists in direction I */
 T ← if TYPE(I) = 'FACE' then
        LC_OT_EQ_FACE_NEIGHBOR(N,I,PATH(P),LEV(P))
```

```
      else if TYPE(I) = 'EDGE' then
        LC_OT_EQ_EDGE_NEIGHBOR(N,I,PATH(P),LEV(P))
      else LC_OT_EQ_VERTEX_NEIGHBOR(N,I,PATH(P),LEV(P));
 if null(T) then return(NIL)/* No neighbor exists in direction I */
 else B ← MAXLEQ(T,L); /* Find maximum locational code in L that is ≤ T */
 if LEV(B) ≥ LEV(P) then return(B)
 else /* The neighbor is smaller; create a new GRAY node */
   begin
    Q ← create(locationalcode);
    PATH(Q) ← T;
    LEV(Q) ← LEV(P);
    COL(Q) ←'GRAY';
    return(Q);
   end;
end;

path array procedure LC_OT_EQ_FACE_NEIGHBOR(N,I,P,LEVEL);
/* Given a 2^N × 2^N × 2^N image represented by a linear octree, return the path from the
    root to the face-neighbor in direction I of a node at level LEVEL whose path from the
    root is P. */
begin
 value integer N;
 value face I;
 value path array P[0 : N − 1];
 value integer LEVEL;
 do
   begin
    P[LEVEL] ← REFLECT(I,P[LEVEL]);
    LEVEL ← LEVEL + 1;
   end
   until LEVEL = N or ADJ(I,P[LEVEL − 1]);
 return(if not ADJ(I,P[LEVEL − 1]) then NIL
        else P);
end;

path array procedure LO_OT_EQ_EDGE_NEIGHBOR(N,I,P,LEVEL);
/* Given a 2^N × 2^N × 2^N image represented by a linear octree, return the path from the
    root to the edge-neighbor in direction I of a node at level LEVEL whose path from the
    root is P. */
begin
 value integer N;
 value edge I;
 value path array P[0 : N − 1];
 value integer LEVEL;
 direction_code PREVIOUS;
 do
   begin
    PREVIOUS ← P[LEVEL];
    P[LEVEL] ← REFLECT(I,PREVIOUS);
    LEVEL ← LEVEL + 1;
   end
   until LEVEL = N or not ADJ(I,PREVIOUS);
 return(if ADJ(I,PREVIOUS) then NIL
        else if COMMON_FACE(I,PREVIOUS) ≠ Ω then
          LC_OT_EQ_FACE_NEIGHBOR(N,COMMON_FACE(I,PREVIOUS),
                                 P,LEVEL)
        else P);
end
```

```
path array procedure LC_OT_EQ_VERTEX_NEIGHBOR(N,I,P,LEVEL);
/* Given a 2^N × 2^N × 2^N image represented by a linear octree, return the path from the
   root to the vertex-neighbor in direction I of a node at level LEVEL whose path from the
   root is P. */
begin
 value integer N;
 value vertex I;
 value path array P[0 : N − 1];
 value integer LEVEL;
 direction_code PREVIOUS;
 do
  begin
   PREVIOUS ← P[LEVEL];
   P[LEVEL] ← REFLECT(I,PREVIOUS);
   LEVEL ← LEVEL + 1;
  end
  until LEVEL = N or not ADJ(I,PREVIOUS);
 return(if ADJ(I,PREVIOUS) then NIL
       else if COMMON_EDGE(I,PREVIOUS) ≠ Ω then
         LC_OT_EQ_EDGE_NEIGHBOR(N,COMMON_EDGE(I,PREVIOUS),
                                P,LEVEL)
       else if COMMON_FACE(I,PREVIOUS) ≠ Ω then
         LC_OT_EQ_FACE_NEIGHBOR(N,COMMON_FACE(I,PREVIOUS),
                                P,LEVEL)
       else P);
end;
```

The process of constructing the locational code of the equal-sized neighbor is analogous to the calculation of the address of the block except that it is more complex since we are dealing with the directional codes rather than the values of the x, y, and z coordinates of the address. In our procedures (e.g., LC_OT_GTEQ_NEIGHBOR, etc.), we have represented the path component of each locational code by an array. Of course, in actuality, we cannot use arrays as they are very wasteful of storage and would defeat the purpose of using a linear octree. Moreover, procedure MAXLEQ, which searches the list of locational codes for the nearest value, is cumbersome when using an array representation of a locational code. Thus, the locational code must be represented as a number. Decoding the individual directional codes is quite easy in this case sine each directional code can be represented by three bits and thus it can be done using shift and mask operations.

6. CONCLUDING REMARKS

We have shown how to find neighbors in all directions of nodes in images represented by octrees. In the case of an octree representation that uses pointers, our algorithms do not make use of coordinate information, knowledge of the size of the image or blocks, or any storage in excess of that imposed by the nature of the octree data structure. When the octree is implemented using a pointerless representation, then the process of neighbor finding is analogous to calculating the address of a neighboring node of equal size, say N, and then searching a list of node addresses to find the smallest block of greater than or equal size that contains N.

It is easy to adapt our algorithms to quadtrees. In this case, we only need to find edge-neighbors and vertex-neighbors. These are accomplished by procedures analogous to OT_EQ_FACE_NEIGHBOR and OT_EQ_EDGE_NEIGHBOR, respec-

tively. In addition, procedure COMMON_EDGE is used in place of procedure COMMON_FACE. Our techniques can also be easily adapted to images of higher dimension. The only requirement is the extension of relations such as ADJ, REFLECT, COMMON_FACE, and COMMON_EDGE, as well as the addition of an appropriate COMMON relation for each additional dimension for the new form of adjacency. Of course, the corresponding tables will also grow considerably larger.

ACKNOWLEDGMENTS

I would like to thank Walid Aref, Jim Purtilo, and MACSYMA for aid in computing the asymptotic costs of the different algorithms. I have also benefitted from the comments of Azriel Rosenfeld. The comments of an anonymous referee are greatly appreciated.

REFERENCES

1. N. Ahuja and C. Nash, Octree representations of moving objects, *Comput. Vision Graphics Image Process.* **26**, No. 2, 1984, 207–216.
2. C. R. Dyer, Computing the Euler number of an image from its quadtree, *Comput. Graphics Image Process.* **13**, No. 3, 1980, 270–276.
3. I. Gargantini, Linear octrees for fast processing of three dimensional objects, *Comput. Graphics Image Process.* **20**, No. 4, 1982, 365–374.
4. I. Gargantini, Detection of connectivity for regions represented by linear quadtrees, *Comput. Math. Appl.* **8**, No. 4, 1982, 319–327.
5. A. S. Glassner, Space subdivision for fast ray tracing, *IEEE Comput. Graphics Appl.* **4**, No. 10, 1984, 15–22.
6. 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.
7. G. M. Hunter and K. Steiglitz, Operations on images using quad trees, *IEEE Trans. Pattern Anal. Mach. Intell.* **1**, No. 2, 1979, 145–153.
8. C. L. Jackins and S. L. Tanimoto, Oct-trees and their use in representing three-dimensional objects, *Comput. Graphics Image Process.* **14**, No. 3, 1980, 249–270.
9. C. L. Jackins and S. L. Tanimoto, Quad-trees, oct-trees, and k-trees—A generalized approach to recursive decomposition of Euclidean space, *IEEE Trans. Pattern Anal. Mach. Intell.* **5**, No. 5, 1983, 533–539.
10. M. Kaplan, The use of spatial coherence in ray tracing, in *Techniques for Computer Graphics* (D. F. Rogers and R. A. Earnshaw, Eds.), pp. 173–193, Springer-Verlag, New York, 1987.
11. E. Kawaguchi and T. Endo, On a method of binary picture representation and its application to data compression, *IEEE Trans. Pattern Anal. Mach. Intell.* **2**, No. 1, 1980, 27–35.
12. A. Klinger, Patterns and search statistics, in *Optimizing Methods in Statistics* (J. S. Rustagi, Ed.), pp. 303–337, Academic Press, New York, 1971.
13. H. Kobayashi, T. Nakamura, and Y. Shigei, Parallel processing of an object space for image synthesis, *Visual Comput.* **3**, No. 1, 1987, 13–22.
14. D. Meagher, Geometric modeling using octree encoding, *Comput. Graphics Image Process.* **19**, No. 2, 1982, 129–147.
15. D. R. Rogers, *Procedural Elements for Computer Graphics*, McGraw–Hill, New York, 1985.
16. H. Samet, Connected component labeling using quadtrees, *J. Assoc. Comput. Mach.* **28**, No. 3, 1981, 487–501.
17. H. Samet, Computing perimeters of images represented by quadtrees, *IEEE Trans. Pattern Anal. Mach. Intell.* **3**, No. 6, 1981, 683–687.
18. H. Samet, Neighbor finding techniques for images represented by quadtrees, *Comput. Graphics Image Process.* **18**, No. 1, 1982, 37–57.
19. H. Samet, The quadtree and related hierarchical data structures, *ACM Comput. Surveys* **16**, No. 2, 1984, 187–260.
20. H. Samet, *Design and Analysis of Spatial Data Structures*, Addison–Wesley, Reading, MA, 1989.
21. H. Samet, *Applications of Spatial Data Structures: Graphics, Image Processing, and GIS*, Addison–Wesley, Reading, MA 1989.
22. H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber, A geographic information system using quadtrees, *Pattern Recognit.* **17**, No. 6, 1984, 647–656.

23. H. Samet and C. A. Shaffer, A model for the analysis of neighbor finding in pointer-based quadtrees, *IEEE Trans. Pattern Anal. Mach. Intell.* **7**, No. 6, 1985, 717–720.
24. H. Samet and R. E. Webber, *A Comparison of the Space Requirements of Multi-dimensional Quadtree-based File Structures*, Computer Science TR-1711, University of Maryland, College Park, MD, September 1986, submitted.
25. H. Samet and R. E. Webber, Hierarchical data structures and algorithms for computer graphics. I. Fundamentals, *IEEE Comput. Graphics Appl.* **8**, No. 3, 1988, 48–68.
26. H. Samet and R. E. Webber, Hierarchical data structures and algorithms for computer graphics. II. Applications, *IEEE Comput. Graphics Appl.* **8**, No. 4, 1988, 59–75.
27. M. Shneier, Calculations of geometric properties using quadtrees, *Comput. Graphics Image Process.* **16**, No. 3, 1981, 296–302.
28. J. Veenstra and N. Ahuja, Octree generation from silhouette views of an object, in *Proceedings, International Conference on Robotics, St. Louis, March 1985*, pp. 843–848.
29. J. Weng and N. Ahuja, Octrees of objects in arbitrary motion: Representation and efficiency, *Comput. Vision Graphics Image Process.* **39**, No. 2, 1987, 167–185.
30. G. Wyvill and T. L. Kunii, A functional model for constructive solid geometry, *Visual Comput.* **1**, No. 1, 1985, 3–14.