BUILDING AN OCTREE FROM A SET OF PARALLELEPIPEDS

Wm. Randolph Franklin
Varol Akman

Electrical, Computer, and Systems Engineering Department
Rensselaer Polytechnic Institute
Troy, New York 12180-3590, USA
518-266-6077

ABSTRACT

We give a novel algorithm for building an octree from a set of parallelepipeds approximating an object. This is an important operation in solid modeling systems based on octrees. The algorithm is simple to program and easy to understand; in fact we give all the code. It creates a minimal octree from the given parallelepipeds. It does not lead to an intermediate storage swell. It is well-suited to handle very precisely specified objects which are made of a large number of parallelepipeds since it can work with linear files which are accessed in an orderly manner to lessen virtual memory page faults.

KEYWORDS: solid modeling, octrees, parallelepiped approximation.

INTRODUCTION

The volume of a solid object, Q, bounded by planar or curved surfaces is easily computed by numerical integration. Q is first approximated by a set of elements bounded by planes (e.g., rectangular parallelepipeds, or PPs for short). These PPs are assumed without loss of generality to be evenly spaced in the xy-plane but to have varying length along the z-axis. Then, the sum of their volumes gives an approximation of the volume of Q. Theoretically, the exact volume of Q is the limit of this sum as the number of PPs approaches to infinity assuming that Q's boundary consists of well-behaving surfaces.

To compute the PPs from Q, one casts parallel rays through the xy-plane [6]. The 2-dimensional spacing, g, of rays in the xy-plane defines two dimensions of the PPs. The third dimension is specified by the entry/exit points of a ray to/from the object. In this paper, we demonstrate the usefulness of parallelepiped approximation in a different context, namely, solid modeling via octrees [1, 3, 4, 5, 10, 12].

Octrees are data structures for modeling solids by symmetric recursive indexing [8]. Assume that Q is inside a cubic universe, W, with edge length $u = 2^{LMAX}$, LMAX integer (typically 10). The universe is divided into u^3 cubes of unit size called voxels. To obtain the octree, Q, W is symmetrically subdivided into eight octants of equal volume. Each of these octants will either be homogeneous (fully occupied by Q or void) or heterogeneous (partially occupied by Q). The heterogeneous octants are further divided into suboctants. This procedure is carried out recursively until octants (possibly single voxels) of uniform properties are obtained. The approximate nature of Q in modeling Q is inherent in the decision step at the voxel level; a partial voxel must either be labeled as full or empty. It is useful to visualize octrees as a generalization of quadtrees [7].

In this paper we give a novel algorithm called STACK for building an octree from a given set of PPs approximating an object. The advantages of STACK are as follows. It is simple to program and easy to understand. It creates a minimal-sized (in a sense to be defined later) octree from the given PPs. It is well-suited to handle very large (i.e., very precisely specified) objects since it can be programmed to work with linear files which are always accessed in an orderly fashion. It does not lead to an intermediate storage swell.

Relevant papers on this subject are quite recent. In [7], a special case, the conversion of 2-dimensional binary arrays to quadtrees has been considered. In [13], an algorithm is given for constructing the tree of a d-dimensional binary image from the trees of its (d - 1)-dimensional cross sections. In [9], an algorithm is given for converting from the boundary representation of a solid to the corresponding octree model utilizing a connected components labeling technique.

DATA STRUCTURES

A set, $s = \{x_1, x_2, \ldots, x_n\}$, is a collection of distinct elements. An interval, [j..k], is a sequence of integers, j, j+1, ..., k. A list, q, is a sequence of elements $[x_1, x_2, \ldots, x_n]$. Element x_1 is the head of q and x_n is the tail. The empty list is denoted by [ ]. There are three fundamental operations on lists:

i) Access: Given a list $q = [x_1, x_2, \ldots, x_n]$ and an integer i, return the i-th element $q(i) = x_i$ of the list.

ii) Sublist: Given a list $q = [x_1, x_2, \ldots, x_n]$ and a pair of integers i and j, return the list $q[i..j] = [x_i, x_{i+1}, \ldots, x_j]$.

iii) Concatenation: Given two lists $q = [x_1, x_2, \ldots, x_n]$ and $r = [y_1, y_2, \ldots, y_m]$, return their concatenation $q.r = [x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m]$. If r has only one element this operation is called append.

We denote the cardinality, n, of a list q by $|q|$. (The same notation is used for sets and for the ordinary absolute value function also.) An n-tuple, $\langle x_1, x_2, \ldots, x_n \rangle$ denotes n elements in that order. In general, the notation of this paper closely follows that of [11].

We start with a description of our input and output data structures, π and Ω, respectively. It is assumed that $u = 2^{LMAX}$ and $g = 2^K$ where $K \in [0..LMAX]$. The elements of π are 4-tuples called PPs:

$$\pi = \{\langle x, y, z_1, z_2 \rangle \mid x, y, z_1, z_2 \in [0..u-1],\ z_1 <= z_2, \text{ and } x, y, z_1, z_2+1 \text{ are all divisible by } g\}.$$

The elements of π will also be denoted by p_i, $i = 1, |\pi|$. The x, y, z_1, and z_2 values of a particular $p \in \pi$ will be denoted by p(x), p(y), $p(z_1)$, and $p(z_2)$, respectively. It is assumed that all PPs in π are mutually disjoint.

We refer the reader to [11] for relevant terminology on trees. In a tree, the level of a node, v, is defined recursively as:

level(v) = 0, if v is the root, and
level(v) = level(f(v)) + 1, otherwise.

Here f(v) denotes the father of v. A node with no sons is a leaf. The level of a tree is understood as the level of its deepest leaf.

The output Ω of our algorithm is an octree (a tree in which every nonleaf node has eight sons) with the following properties:

i) the nodes of Ω are labeled with three types: empty, full, and partial,

ii) the root of Ω is always partial except when π is trivially equal to a completely full (resp. completely empty) W in which case it becomes full (resp. empty),

iii) the level of Ω is LMAX' = log u - log g = LMAX - K (In this paper log always denotes $\log_2$),

iv) the leaves of Ω are either empty or full, and

v) the nonleaf nodes of Ω are partial.

Before we describe our main data structure, we give a few definitions to make the upcoming algorithmic description easier. A row at level i is a 3-tuple $\langle x, y, z \rangle$ where z is divisible by $h = 2^{LMAX-i}$; this is a shorthand for PP $\langle x, y, z, z_2 \rangle$ where $z_2 = z + h - 1$. It is noted that the z-length of a row at level i is always h units or h/g spacings. Two rows $r_1 = \langle x_1, y_1, z \rangle$ and $r_2 = \langle x_2, y_2, z \rangle$ at the same level are called adjacent if $x_1 = x_2$ and $|y_1 - y_2| = g$. (Note that this definition requires that they have the same z-length.) 2^i ($i \in [1..LMAX]$) rows at level LMAX - i are combinable if when sorted in y to be $r_1, r_2, \ldots$ then every intermediate r_j in this sequence is adjacent to its predecessor and successor.

For example, the rows $\langle 0, 0, 0 \rangle$, $\langle 0, 1, 0 \rangle$, $\langle 0, 2, 0 \rangle$, and $\langle 0, 3, 0 \rangle$ at level LMAX - 1 are combinable while the rows $\langle 0, 1, 0 \rangle$, $\langle 1, 1, 0 \rangle$, $\langle 2, 1, 0 \rangle$, and $\langle 2, 0, 0 \rangle$ at level LMAX - 1 are not.

Let $r_1, r_2, \ldots$ be 2^i combinable rows at level LMAX - i. A square, s, at level LMAX - i of 2^{LMAX-i} by 2^{LMAX-i} by 1 voxels is obtained by combining them into a single 3-tuple $\langle x, y, z \rangle$ where $s(y) = \min_j(r_j(y))$, and $s(x) = r_1(x)$ and $s(z) = r_1(z)$. Two squares $s_1 = \langle x_1, y_1, z \rangle$ and $s_2 = \langle x_2, y_2, z \rangle$ at the same level are called to be adjacent if $y_1 = y_2$ and $|x_1 - x_2| = g$. (They have the same z-length.) 2^i ($i \in [1..LMAX]$) squares are combinable if when sorted in x to be $s_1, s_2, \ldots$ then every s_j in this sequence is adjacent to its predecessor and successor.

For example, the squares $\langle 0, 0, 0 \rangle$ and $\langle 1, 0, 0 \rangle$ at level LMAX - 1 are combinable while the squares $\langle 0, 0, 0 \rangle$ and $\langle 0, 3, 0 \rangle$ at level LMAX - 1 are not.

Let $s_1, s_2, \ldots$ be 2^i combinable squares at level LMAX - i. A cube, c, at level LMAX - i of 2^{LMAX-i} by 2^{LMAX-i} by 2^{LMAX-i} voxels is obtained from their combination as a 3-tuple ⟨x, y, z⟩ where $c(x) = \min_j(r_j(x))$, and $c(y) = r_1(y)$ and $c(z) = r_1(z)$.

If a row, ⟨x, y, z⟩, at level i, i < LMAX, is split in the z direction then two rows, ⟨x, y, z⟩ and ⟨x, y, z + h⟩, are obtained at level i + 1. If a square, ⟨x, y, z⟩, at level i is split in x and y directions then four squares, ⟨x, y, z⟩, ⟨x, y + h, z⟩, ⟨x + h, y, z⟩, ⟨x + h, y + h, z⟩, are obtained at level i + 1. In both cases, $h = 2^{LMAX-i-1}$. It should be clear that the idea of splitting is generalizable to cubes and hypercubes.

The maximal components of a PP p form a list $[m_1, m_2, \ldots]$ of rows where each m_i is a component. To find the components, first search for the longest (in z) row in p. This is a component. Remove it from p. This either reduces p to a shorter (in z) PP or partitions it into two PPs which are also shorter than p. In any case, this procedure recurses until a created component has z-length g. In this case it is not further partitioned. It is noted that, once the maximal components are found it should be impossible to obtain a longer component by combining two components.

For example, the maximal components of the PP ⟨1, 1, 17, 93⟩ are the list of rows [⟨1, 1, 17⟩ at level LMAX, ⟨1, 1, 18⟩ at level LMAX - 1, ⟨1, 1, 20⟩ at level LMAX - 2, ⟨1, 1, 24⟩ at level LMAX - 3, ⟨1, 1, 32⟩ at level LMAX - 5, ⟨1, 1, 64⟩ at level LMAX - 4, ⟨1, 1, 80⟩ at level LMAX - 3, ⟨1, 1, 88⟩ at level LMAX - 2, ⟨1, 1, 92⟩ at level LMAX].

Our main data structure consists of a set of at most DMAX(LMAX' + 1) - 1 lists that we will call δλ-lists (dimension-level lists). Here, DMAX is the maximum dimension of W and LMAX' = LMAX - K, as before. A δλ-list at dimension D and level L is denoted as $t_{D,L}$. There are LMAX' 1-dimensional δλ-lists, LMAX' 2-dimensional δλ-lists, and LMAX' - 1 3-dimensional δλ-lists when D = 3. (In general, the number of the highest dimensional lists will be one less than their predecessors.) The elements of $t_{D,L}$ are rows if D = 1, squares if D = 2, cubes if D = 3, and hypercubes if D > 3. Although our algorithm will still be correct for D > 3, we will not be concerned with this anymore since its practical value is questionable in the absence of affordable 4-D display devices.

When $|\pi|$ is very large it may be advantageous to employ linear disk files to hold the δλ-lists. In this case, only three files will be open during the execution of our algorithm: $t_{D,L}$ for read and $t_{D+1,L}$ and $t_{D,L+1}$ both for write. Since reads always take place sequentially and writes are always carried out as appends the algorithm is on solid ground against virtual memory page faults.

Finally, although we have a language with dynamic data structuring facilities in mind to implement this algorithm, for static languages (such as Fortran) a list space to hold 2(LMAX' + 1) δλ-lists would be enough for any DMAX > 2. This is due to the fact that once the combine/split operation (to be explained later) is finished with 1-dimensional δλ-lists one can allocate for the 3-dimensional lists the same space occupied by them, and so on.

ALGORITHM

In the following, to express our algorithm, an Algol-like language combining Dijkstra's guarded command language and SETL is used. This language is described in [11] in detail and will not be explained here.

Throughout this paper DMAX will denote the maximum dimension which is typically 3; D is the current dimension. LMAX denotes the maximum level which is typically 10 for a spacing value g = 1; L is the current level. The universe, W, is at level 0 and an LMAX-level full octree has 8^{LMAX} lowest level nodes. Using a larger spacing it is possible to reduce the maximum level to LMAX' = LMAX - log g.

A brief summary of our algorithm, STACK, is as follows. First, STACK tries to combine adjacent rows into squares. (Assume that, each PP has been divided into its maximal components and these have already been inserted into relevant 1-dimensional δλ-lists using MAXCOM below.) If a row cannot be combined then it is split into two smaller (half-size) rows and they are tried, until the remaining pieces are at level LMAX'. These are inserted into Ω since there is no way to combine them.

Then, STACK tries to combine adjacent squares into cubes. Any square that cannot be combined is split into four smaller (quarter-size) squares and the process is repeated until the remaining pieces are at level LMAX', and they are added to Ω. Finally, all the cubes that were produced are added to Ω. We will show in the next section that this builds Ω in its reduced form. (An octree is in reduced form if it has no partial nodes having all empty or all full sons.)

In the following we give the main program and the other components of STACK. (This name is chosen to conjure up a vision of what this algorithm is doing, i.e., stacking up things to build larger things.)

```
PROC stack(SET π, TREE Ω);
COMMENT main procedure to create an octree
        from a set of PPs;
INTEGER L, D, g, K, LMAX, LMAX';
TUPLE p;
LIST t_{D,L};
COMMENT initialize (assume that LMAX := 10
        and K := log(g));
LMAX' := LMAX - K;
FOR D ε [1..3] ->
 FOR L ε [0..LMAX'] ->
  t_{D,L} := [|
 ROF
ROF;
Ω := NULL;
COMMENT read π and insert its maximal components
        into 1-D δλ-lists;
FOR p ε π -> maxcom(p, 0, 2^LMAX - 1, 0) ROF;
COMMENT start combine/split operation;
FOR D ε [1..2] ->
 FOR L ε [0..LMAX'-1] ->
  IF D = 1 -> SORT t_{D,L} BY y,z,x; csrow(L)
   | D = 2 -> SORT t_{D,L} BY z,x,y; cssqr(L)
  FI
 ROF;
 add elements of t_{D,LMAX'} to Ω
ROF;
FOR L ε [0..LMAX'-1] ->
 add elements of t_{3,L} to Ω
ROF;
COMMENT at this point Ω is obtained;
RETURN
END stack;
```

```
PROC maxcom(TUPLE p, INTEGER lo, hi, L,
            MODIFIES LIST t_{1,L});
COMMENT find and add maximal components of p
        to 1-dimensional δλ-lists;
COMMENT lo and hi are the initial bounds of
        a maximal component.
INTEGER nlo, nhi, tmp;
COMMENT nlo and nhi are the running bounds of
        a maximal component.
IF p(z_1) = lo AND p(z_2) = hi -> t_{1,L} := t_{1,L}.[p]
 | p(z_1) >= lo AND p(z_2) <= hi ->
   L := L + 1;
   nhi := (hi + lo + 1)/2 - 1;
   nlo := nhi + 1;
   tmp := L;
   IF p(z_1) <= nhi AND p(z_2) <= nhi ->
      maxcom(p, lo, nhi, L)
    | p(z_1) >= nlo AND p(z_2) >= nlo ->
      maxcom(p, nlo, hi, L)
    | p(z_1) <= nhi AND p(z_2) >= nlo ->
      maxcom(<x, y, z_1, nhi>, lo, nhi, L);
      L := tmp;
      maxcom(<x, y, nlo, z_2>, nlo, hi, L)
   FI;
   L := tmp
FI;
RETURN
END maxcom;
```

```
PROC csrow(INTEGER L,
           MODIFIES LISTS t1,L, t2,L, t1,L+1);
COMMENT combine/split δλ-list t1,L;
INTEGER i, j , e, he, n, D;
TUPLE r, s, q;
e := 2^(LMAX-L); he := e/2; D := 1; n := |tD,L|;
i := 0;
DO UNTIL i = n ->
 i := i + 1;
 COMMENT let r = ⟨x1, y1, z1⟩ be the i-th
         element of tD,L;
 IF mod(r(x), e) <> 0 ->
    tD,L+1 := tD,L+1.[r, ⟨x1, y1, z1+he⟩]
  | mod(r(x), e) = 0 ->
    j := i + e/g - 1;
    IF j > n ->
     FOR m ε [i..n] ->
      COMMENT let s = ⟨x, y, z⟩ be the m-th
              element of tD,L;
      tD,L+1 := tD,L+1.[s, ⟨x, y, z+he⟩]
     ROF;
     BREAK
    FI;
    COMMENT let q = ⟨x2, y2, z2⟩ be the j-th
            element of tD,L;
    IF q(y) <> r(y) OR q(z) <> r(z) ->
       tD,L+1 := tD,L+1.[r, ⟨x1, y1, z1+he⟩]
     | q(y) = r(y) AND q(z) = r(z) ->
       COMMENT combine;
       tD+1,L := tD+1,L.[r]; i := j
    FI
 FI
OD;
tD,L := [];
RETURN
END csrow;
```

```
PROC cssqr(INTEGER L,
           MODIFIES LISTS t2,L, t3,L, t2,L+1);
COMMENT combine/split δλ-list t2,L;
INTEGER i, j , e, he, n, D;
TUPLE r, s, q;
e := 2^(LMAX-L); he := e/2; D := 2; n := |tD,L|;
i := 0;
DO UNTIL i = n ->
 i := i + 1;
 COMMENT let r = ⟨x1, y1, z1⟩ be the i-th
         element of tD,L;
 IF mod(r(y), e) <> 0 ->
    tD,L+1 := tD,L+1.[r, ⟨x1, y1, z1+he⟩,
                        ⟨x1+he, y1, z1⟩,
                        ⟨x1+he, y1, z1+he⟩]
  | mod(r(y), e) = 0 ->
    j := i + e/g - 1;
    IF j > n ->
     FOR m ε [i..n] ->
      COMMENT let s = ⟨x, y, z⟩ be the m-th
              element of tD,L;
      tD,L+1 := tD,L+1.[s, ⟨x, y, z+he⟩,
                          ⟨x+he, y, z⟩,
                          ⟨x+he, y, z+he⟩]
     ROF;
     BREAK
    FI;
    COMMENT let q = ⟨x2, y2, z2⟩ be the j-th
            element of tD,L;
    IF q(x) <> r(x) OR q(z) <> r(z) ->
       tD,L+1 := tD,L+1.[r, ⟨x1, y1, z1+he⟩,
                           ⟨x1+he, y1, z1⟩,
                           ⟨x1+he, y1, z1+he⟩]
     | q(x) = r(x) AND q(z) = r(z) ->
       COMMENT combine;
       tD+1,L := tD+1,L.[r]; i := j
    FI
 FI
OD;
tD,L := [];
RETURN
END cssqr;
```

In STACK, the high-level operation "SORT list BY key" is lexicographic since key is composite. In the same procedure, the "addition of a full node to Ω" is intentionally left as a high-level step. This is due to the fact that an octree is basically a digital search tree (also known as trie) and handling insertion in a trie is well-known [2].

We state several properties of STACK deduced from these procedures.

Lemma 1: level(Ω) <= LMAX'.
Proof: Obvious since the minimum cube must have an edge length >= g.

Lemma 2: The elements of $t_{1,LMAX'}$ and $t_{2,LMAX'}$ cannot be combined and hence are full nodes of Ω.
Proof: Trivial.

Lemma 3: There is no need for $t_{3,LMAX'}$.
Proof: Any input to $t_{3,LMAX'}$ may come only from $t_{2,LMAX'}$ which is [] at that point. Additionally, the latter cannot send the former anything since it cannot combine due to Lemma 2.

Lemma 4: Ω is always in reduced form after STACK is applied.
Proof: Assume that this is not true. Take any partial node of Ω at level L which has eight full nodes. (Eight empty nodes are treated similarly.) These certainly imply 2^{LMAX-L} combinable squares at level L + 1 which must have been correctly computed by CSROW procedure. But then CSSQR would correctly combine them to a full cube at level L.

EFFICIENCY

To estimate the efficiency of STACK we examine its individual steps. Since we are trying to see the worst-case complexity assume that g = 1, thus LMAX' = LMAX.

For a given PP there may be as many as 2(LMAX - 1) maximal components. Therefore, MAXCOM initializes all the 1-dimensional δλ-lists with rows in O(LMAX $|\pi|$) operations under the assumption that appends take O(1) time.

Sorting a δλ-list is a common operation in STACK. The important point is that for D > 1, lists $t_{D,L}$ will not be completely scrambled prior to sorting. Because of the way that new elements are appended into them in almost sorted order, they will have some order in them. (We refer the reader to CSROW and CSSQR to see this clearly.) On the other hand, one can assume that there will be no order in 1-dimensional δλ-lists initially; they are in random order. This would not be true if the elements of π are listed in some order; this may happen if the ray-casting is implemented in some methodical manner such as via do-loops while computing the PPs. It is also noted that 1-dimensional splits introduce some order to 1-dimensional δλ-lists also. To exploit the last fact one can use Shell sort which is of average-case $O(n^{1.25})$. It is known that Shell sort has worst-case of $O(n^{1.5})$ and furthermore does less work when the file is partially ordered [2].

Finally, it is emphasized that after the sorting step, CSROW and CSSQR execute very efficiently since they make a single pass over the list and spend $|t_{D,L}|$ time since appends are carried out in constant time.

IMPLEMENTATION RESULTS

We implemented STACK in Ratfor (a structured dialect of Fortran). For a 1/8-sphere, the elapsed CPU time of the algorithm is 9.2 seconds on a Prime 750. This object is built from 833 PPs with LMAX = 10 and g = 16. The final octree has a total of 6569 nodes (4090 full, 1664 full with surface normals -- see the explanation of surface normals below). For a paraboloid built from 916 PPs with LMAX = 10 and g = 32 the final octree has a total of 5913 nodes (3248 full, 1832 full with surface normals). This takes 7.4 seconds of CPU time. In agreement with our predictions, the I/O time is low in both cases (0.9 and 0.3 seconds, respectively). For a precisely specified 1/8-sphere consisting of 12985 PPs, STACK takes about 3 CPU minutes to build the final octree which has 106833 nodes and LMAX = 8. The node distribution is 67570 full nodes, 13354 partial nodes, and 25909 empty nodes. This object is larger than many of the examples cited in [9] and [13].

In the sequel we describe an enhancement (which we also implemented) of this algorithm.

Since an octree created by STACK must eventually be displayed, most of the time PPs will also have surface normal vectors, n_1 and n_2, associated with their z_1 and z_2 endpoints, respectively. That is, p is a 6-tuple $\langle x, y, z_1, z_2, n_1, n_2 \rangle$ where:

$n_1 = n_1(x)i + n_1(y)j + n_1(z)k$, and
$n_2 = n_2(x)i + n_2(y)j + n_2(z)k$. (Here, i, j, and k are the unit vectors in x, y, and z directions, respectively.)

In this case, to create Ω from π, the following approach may be used. Create for each $p \in \pi$ three PPs p_1, p_2, and p' where:

$p_1 = \langle x, y, z_1, z_1 + g - 1 \rangle$ with implied normal n_1,
$p_2 = \langle x, y, z_2 - g + 1, z_2 \rangle$ with implied normal n_2, and
$p' = \langle x, y, z_1 + g, z_2 - g \rangle$ with no normals,

assuming that $p = \langle x, y, z_1, z_2, n_1, n_2 \rangle$, $z_2 - z_1 > g - 1$. (If for a particular p, $z_2 - z_1 = g - 1$ then only p_1 is created with implied normal n_1. This happened twice in the above 1/8-sphere as can be seen from the number of full nodes with normals.) Once this partitioning is done, the idea is to add p_1 and p_2 along with their normals to Ω directly since these must not be combined. Then for π' (which is the set including all p') STACK is applied as before. Basically, what we are doing can be summarized as "peeling off the skin" of π to obtain π'.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grant number ECS 83-51942 and the Office of Naval Research, Information Sciences Division, contract number N00014-82-K-0301. This information does not necessarily reflect the position of the government, and no official endorsement should be inferred. The second author is also supported in part by a Fulbright award.

REFERENCES

1. DOCTOR, L.J., AND TORBORG, J.G. 1981. Display techniques for octree-encoded objects. IEEE Computer Graphics and Applications 1, 3, 29-38.

2. GONNET, G.H. 1984. Handbook of Algorithms and Data Structures, International Computer Science Series, Addison-Wesley, Reading, Mass.

3. JACKINS, C.L., AND TANIMOTO, S.L. 1980. Octrees and their use in representing three-dimensional objects. Computer Graphics and Image Processing 14, 249-270.

4. JACKINS, C.L., AND TANIMOTO, S.L. 1983. Quadtrees, octrees, and K-trees: a generalized approach to recursive decomposition of Euclidean space. IEEE Transactions on Pattern Analysis and Machine Intelligence 5, 5, 533-539.

5. MEAGHER, D. 1982. Geometric modeling using octree encoding. Computer Graphics and Image Processing 19, 129-147.

6. ROTH, S.D. 1980. Ray casting as a method for solid modeling. Tech. Rep. GMR-3466, Computer Science Dept., General Motors Research Labs, Warren, Mich.

7. SAMET, H. 1980. Region representation: quadtrees from boundary codes. Communications of the ACM 23, 3, 163-170.

8. SRIHARI, S.N. 1981. Representation of three-dimensional digital images. ACM Computing Surveys 13, 4, 400-424.

9. TAMMINEN, M., AND SAMET, H. 1984. Efficient octree conversion by connectivity labeling. SIGGRAPH'84 Proceedings (published as ACM Computer Graphics 18, 3), 43-51.

10. TANIMOTO, S. 1980. Image data structures. In S. Tanimoto and A. Klinger (Eds.), Structured Computer Vision, Academic Press, New York.

11. TARJAN, R.E. 1983. Data Structures and Network Algorithms, CBMS-NSF Regional Conference Series in Applied Math. 44, SIAM, Philadelphia, Pa.

12. YAMAGUCHI, K., KUNII, T.L., FUJIMURA, K., AND TORIYA, H. 1984. Octree-related data structures and algorithms. IEEE Computer Graphics and Applications 4, 1, 53-59.

13. YAU, M., AND SRIHARI, S.N. 1983. A hierarchical data structure for multidimensional images. Communications of the ACM 26, 7, 504-515.