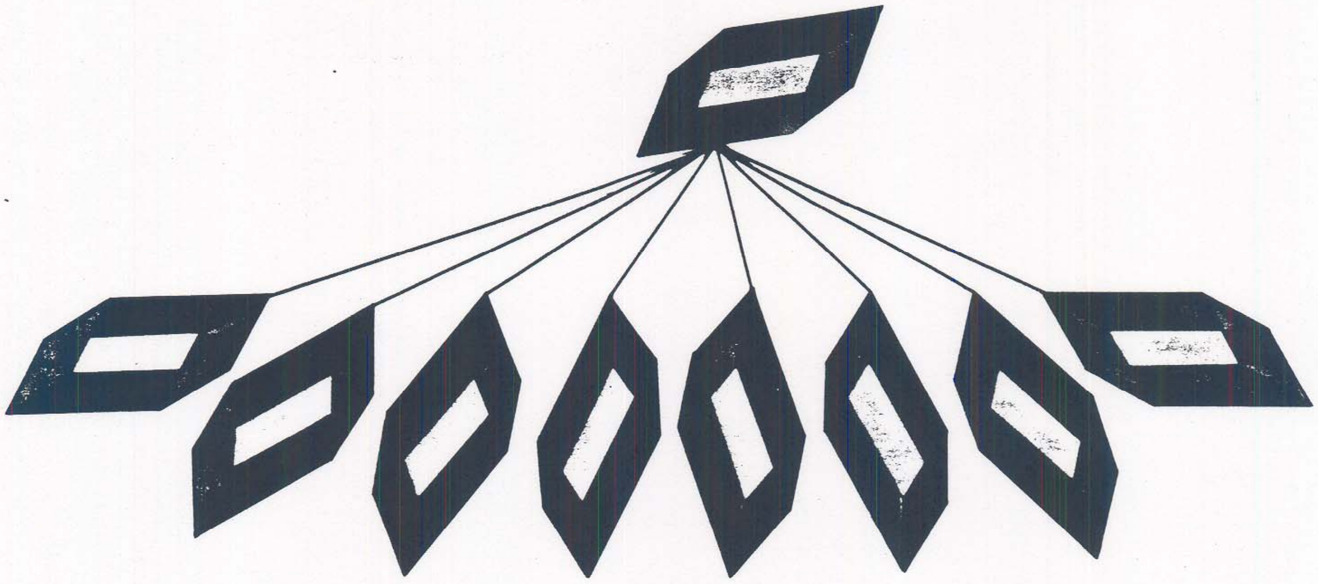
40

What do you call an algorithm designed to stack things up to build larger things? Why, STACK, of course.

Building an Octree from a Set of Parallelepipeds



Wm. Randolph Franklin and Varol Akman

Rensselaer Polytechnic Institute

In this article we provide a [illegible] 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 [illegible] from the given parallelepipeds and does not lead to intermediate storage swell. It is well suited for handling precisely specified objects made from a large number of parallelepipeds, since it only works with [illegible] which are accessed [illegible] virtual-memory page faults.

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 lengths 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 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.[1] The two-dimensional spacing g of rays in the xy-plane defines two dimensions of the PPs.

0272-1716/85/1000-0058$01.00 © 1985 IEEE

The third dimension is specified by the entry/exit points of a ray to/from the object. In this article we demonstrate the usefulness of parallelepiped approximation in a different context: namely, solid modeling via octrees.[2-7]

Octrees are data structures for modeling solids by symmetric recursive indexing.[8] Assume that Q is inside a cubic universe W with edge length $u = 2^{LMAX}$, $LMAX$ an integer (typically 10). The universe is divided into u^3 cubes of a unit size called voxels. To obtain the octree Ω, W is symmetrically subdivided into eight octants of equal volume. Each of these octants will be either 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 Ω in modeling Q is inherent in the decision step at the voxel level; a partial voxel must be labeled either full or empty. It is useful to visualize octrees as a generalization of quadtrees.[9]

In this article we provide a novel algorithm called STACK for building an octree from a given set of PPs approximating an object. The advantages of STACK are that it is simple to program and understand, it creates a minimal-sized octree (in a sense to be defined later) from the given PPs, and it is well-suited for handling 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. Furthermore, it does not lead to an intermediate storage swell.

Relevant papers on this subject are quite recent. Samet considers a special case, the conversion of two-dimensional binary arrays to quadtrees.[9] Yau and Srihari give an algorithm for constructing the tree of a d-dimensional binary image from the trees of its $(d-1)$-dimensional cross sections.[10] Tamminen and Samet give an algorithm for converting from the boundary representation of a solid to the corresponding octree model using a connected components labeling technique.[11]

Data structures

A set $s = \{x_1, x_2, \ldots, x_n\}$ is a collection of distinct elements. An interval $[j \ldots k]$ is a sequence of integers, j, $j+1, \ldots, k$. 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:

1. Access: Given a list $q = [x_1, x_2, \ldots, x_n]$ and an integer i, return the ith element $q(i) = x_i$ of the list.
2. Sublist: Given a list $q = [x_1, x_2, \ldots, x_n]$ and a pair of integers i and j, return the list $q[i \ldots j] = [x_i, x_{i+1}, \ldots, x_j]$.
3. 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 article closely follows that of Tarjan.[12]

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 $K \epsilon [0 \ldots LMAX]$. The elements of π are 4-tuples called PPs:

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

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 $p \epsilon \pi$ 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 Tarjan[12] for relevant terminology on trees. In a tree, the level of a node v is defined recursively as

$$\text{level}(v) = 0, \text{ if } v \text{ is the root, and}$$
$$\text{level}(v) = \text{level}(f(v)) + 1, \text{ otherwise.}$$

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:

1. The nodes of Ω are labeled with three types: empty, full, and partial.
2. The root of Ω is always partial except when π is trivially equal to a completely full (respectively completely empty) W, in which case it becomes full (respectively empty).
3. The level of Ω is $LMAX' = \log u - \log g = LMAX - K$. (In this article log always denotes $\log_2$.)
4. The leaves of Ω are either empty or full.
5. 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 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 $|y_1 - y_2| = g$. (This definition requires that they have the same z-length.) 2^i ($i \epsilon [1 \ldots LMAX]$) rows at level $LMAX - i$ are combinable if, when sorted in y to be $r_1, r_2, \ldots$, every intermediate r_j in this sequence is adjacent to its predecessor and successor and r_1 is a multiple of h.

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

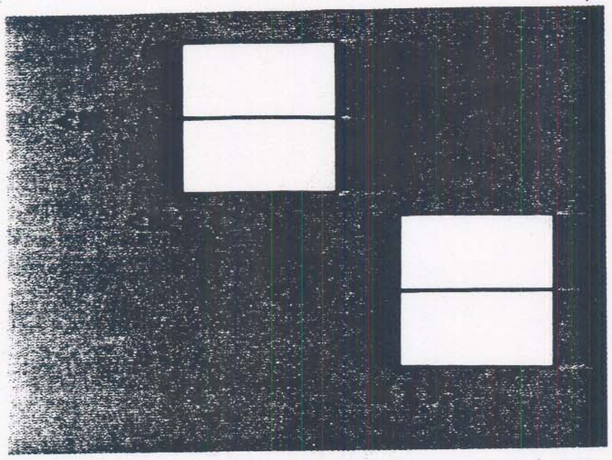


Figure 1. Two rows of length $h = 2$ that are combinable into one square (a). Two rows that cannot be combined, since they are misaligned (b).

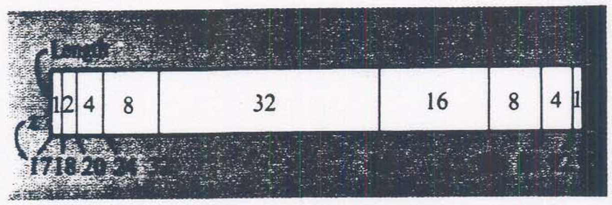


Figure 2. Splitting a PP <1, 1, 17, 93> into nine rows.

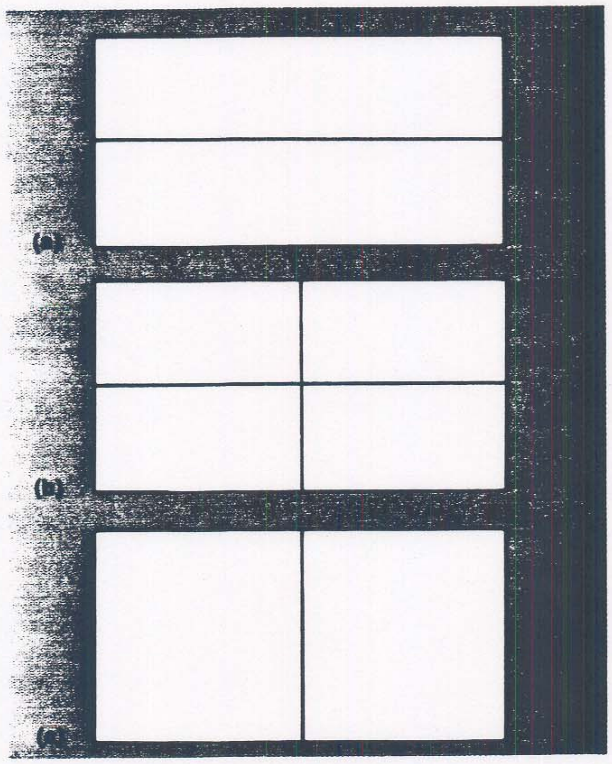


Figure 3. Two rows of length 4 (a) that cannot be combined into a square but can be split into four rows of length 2 (b) and then combined into two squares of side 2 (c).

the rows $<0, 1, 0>$, $<1, 1, 0>$, $<2, 0, 0>$, and $<2, 1, 0>$ at level $LMAX - 1$ are not, since they are not adjacent. (See Figure 1 for other examples.)

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 $<x, y, z>$ where $s(y) = \min_j (r_j(y))$, and $s(x) = r_1(x)$, and $s(z) = r_1(z)$. Two squares $s_1 = <x_1, y_1, z>$ and $s_2 = <x_2, y_2, z>$ at the same level are called adjacent if $y_1 = y_2$ and $|x_1 - x_2| = g$. (They have the same z-length.) 2^i ($i \in [1 \ldots LMAX]$) squares are combinable if when sorted in x to be $s_1, s_2, \ldots$ every s_j in this sequence is adjacent to its predecessor and successor, and s_1 is a multiple of h.

For example, the squares $<0, 0, 0>$ and $<1, 0, 0>$ at level $LMAX - 1$ are combinable, while the squares $<0, 0, 0>$ and $<0, 3, 0>$ 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, $<x, y, z>$, $<x, y + h, z>$, $<x + h, y, z>$, and $<x + h, y + h, z>$, are obtained at level $i + 1$. In both cases $h = 2^{LMAX-i-1}$. It should be clear that the idea of splitting can be generalized to cubes and hypercubes.

The maximal components of a single PP p form a list $[m_1, m_2, \ldots]$ of rows in which 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. It is then partitioned no further. 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 $LMAX - 2$, and $<1, 1, 92>$ at level $LMAX$], as shown in Figure 2.

Our main data structure consists of a set of at most $DMAX(LMAX' + 1) - 1$ lists that we will call $\delta\lambda$-lists (dimension-level lists). Here, $DMAX$ is the maximum dimension of W, and $LMAX' = LMAX - K$, as before. A $\delta\lambda$-list at dimension D and level L is denoted as $t_{D,L}$. There are $LMAX'$ one-dimensional $\delta\lambda$-lists, $LMAX'$ two-dimensional $\delta\lambda$-lists, and $LMAX' - 1$ three-dimensional $\delta\lambda$-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 $\delta\lambda$-lists. In this case only three files will be open during the execution of our algorithm: $t_{D,L}$ for read, and both $t_{D+1,L}$ and $t_{D,L+1}$ 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 $2(LMAX' + 1)$ $\delta\lambda$-lists would be enough for any $DMAX > 2$. This is because once the combine/split operation (to be explained later) is finished with one-dimensional $\delta\lambda$-lists, one can allocate the space they occupied for the three-dimensional lists, and so on.

Algorithm

The following uses an Algol-like language combining Dijkstra's guarded command language and SETL to express our algorithm. This language is described in detail by Tarjan[12] and will not be explained here.

Throughout this article $DMAX$ 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, shows that STACK first 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 one-dimensional $\delta\lambda$-lists using MAXCOM below.) If a row cannot be combined, it is split into two smaller (half-size) rows that are tried until the remaining pieces are at level $LMAX'$. (See Figure 3.) 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 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 because an octree is basically a digital search tree (also known as trie), and handling insertion in a trie is well-known.[13]

Several properties of STACK can be deduced from these procedures:

Lemma 1: Level $(\Omega) \leq LMAX'$.
Proof: Obvious, since the minimum cube must have an edge length $\geq g$.

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 $t_{3,LMAX'}$ can come only from $t_{2,LMAX'}$, which is [] at that point. Additionally, the latter cannot send the former anything, since it cannot combine because of 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 that 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 the CSROW procedure. But then CSSQR would correctly combine them into 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, let's assume that $g = 1$; thus $LMAX' = LMAX$.

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

Sorting a $\delta\lambda$-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 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 one-dimensional $\delta\lambda$-lists initially; they are in random order. This would not be true if the elements of π were listed in some order; this may happen if the ray-casting is implemented in some methodical manner, for instance via do-loops while computing the PPs. It is noted that one-dimensional splits also introduce some order to one-dimensional $\delta\lambda$-lists. To exploit this last fact, one can use Shell sort, which is of average-case $O(n^{1.25})$.[13] A quick sort routine that handles partly sorted lists efficiently would also be excellent.

Finally, we emphasize 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. (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,

In this box we give the main program and the [illegible] (STACK). This name was chosen to conjure up a vision of [illegible]

```
PROC [illegible]                          [illegible] INTEGER [illegible]
COMMENT [illegible]                       [illegible] LIST [illegible]
                                          [illegible] and add maximal components of [illegible]
INTEGER [illegible]                       [illegible] $\delta\lambda$-lists
TUPLE [illegible]                         [illegible] are the initial bounds of
LIST [illegible]                          [illegible] components
COMMENT [illegible]                       INTEGER [illegible]
                                          [illegible] are the running bounds of
LMAX ← [illegible]                        [illegible] components
FOR D [illegible]                         [illegible]
   FOR L [illegible]                      [illegible]
      [illegible]
   ROF
ROF
Ω ← NULL [illegible]
COMMENT [illegible]                       [illegible]
FOR p [illegible]                         [illegible]
COMMENT [illegible]                       [illegible]
FOR D [illegible]                         [illegible]
   FOR L [illegible]                      [illegible]
      IF D [illegible]                    [illegible]
         [illegible]                      [illegible]
      FI
   ROF
   add [illegible]
ROF
FOR [illegible]
   add [illegible]
ROF
COMMENT [illegible]
RETURN [illegible]
END stack
```

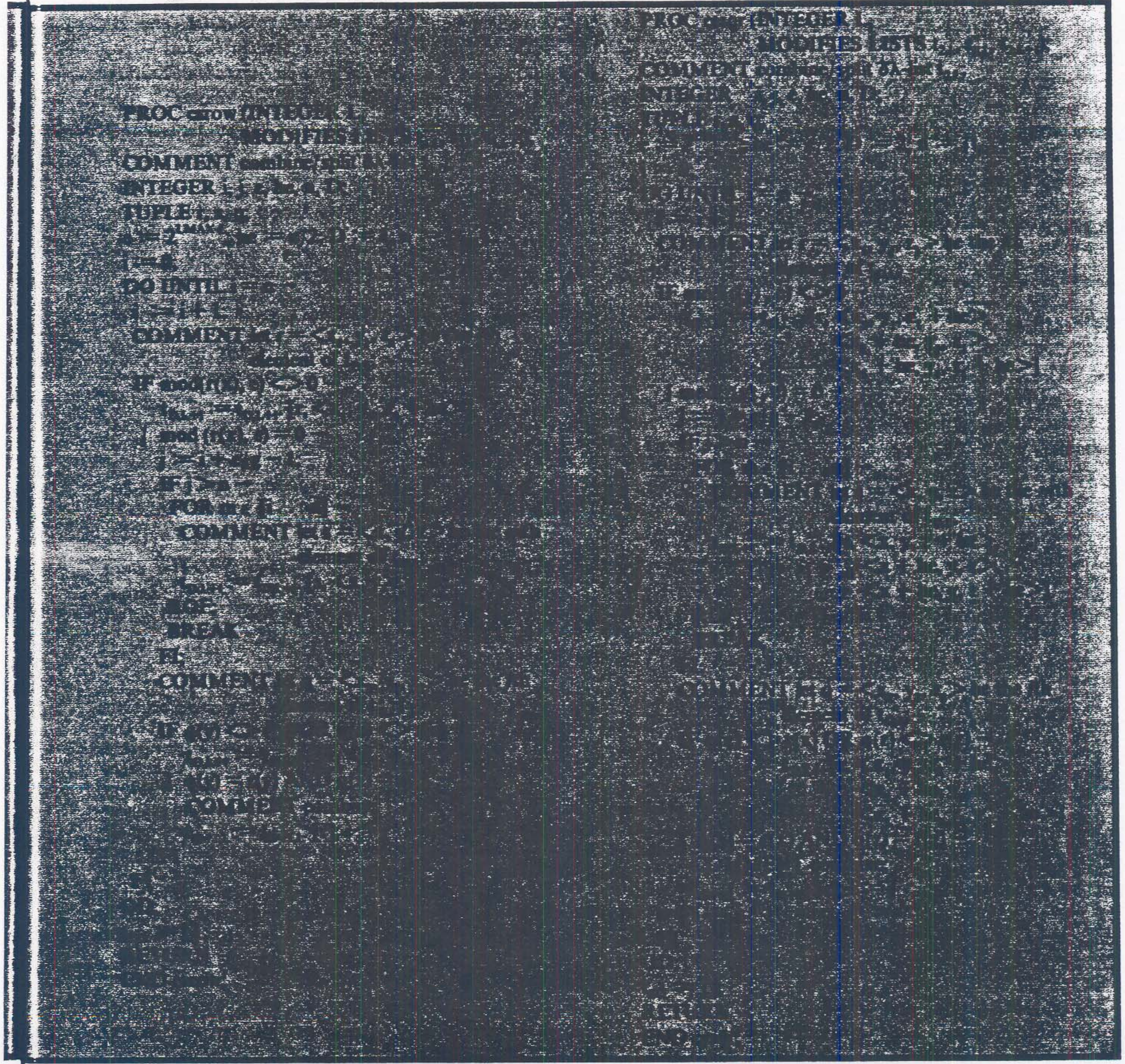
the I/O time, which is due to paging, is low in both cases (0.9 and 0.3 seconds, respectively). For a precisely specified 1/8 sphere consisting of 12,985 PPs, STACK takes about 3 CPU minutes to build the final octree, which has 106,833 nodes and $LMAX = 8$. The node distribution is 67,570 full nodes, 13,354 partial nodes, and 25,909 empty nodes. This object is larger than many of the examples cited by Yau and Srihari[10] and by Tamminen and Samet.[11]

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 $<x, y, z_1, z_2, n_1, n_2>$ 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 can be used. Create for each $p \in \pi$ three PPs p_1, p_2, and p' where

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



```
PROC grow (INTEGER L, [illegible]
          MODIFIES [illegible]
COMMENT combine/split [illegible]
INTEGER [illegible]
TUPLE [illegible]
[illegible] := 2^{LMAX} [illegible]
[illegible]
DO UNTIL [illegible]
  [illegible]
  COMMENT [illegible]
          [illegible]
  IF mod( [illegible] ) <> 0 [illegible]
    [illegible]
    mod ( [illegible] ) = 0
    [illegible]
    IF [illegible]
      FOR [illegible]
        COMMENT [illegible]
        [illegible]
    ELSE
      BREAK
    FI
    COMMENT [illegible]
    IF [illegible]
      [illegible]
      COMMENT [illegible]
[illegible]
RETURN
[illegible]

PROC [illegible] (INTEGER L,
          MODIFIES LISTS [illegible]
COMMENT [illegible]
INTEGER [illegible]
TUPLE [illegible]
[illegible]
[illegible]
  COMMENT [illegible]
  IF [illegible]
  [illegible]
    COMMENT [illegible]
  [illegible]
  COMMENT [illegible]
[illegible]
COMMENT
RETURN
END
```

assuming that $p = <x, y, z_1, z_2, n_1, n_2>, z_2 - z_1 > g - 1$. (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 on 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 Schlumberger-Doll Research Corp., Ridgefield, Connecticut, also provided partial support for the research. In addition, the second author is supported in part by a Fulbright award.

References

1. S. D. Roth, "Ray Casting as a Method for Solid Modeling," tech. report GMR-3466, Computer Science Dept., General Motors Research Labs, Warren, MI, 1980.
2. L. J. Doctor and J. G. Torborg, "Display Techniques for Octree-Encoded Objects," *IEEE Computer Graphics and Applications,* Vol. 1, No. 3, July 1981, pp. 29-38.
3. C. L. Jackins and S. L. Tanimoto, "Octrees and Their Use in Representing Three-Dimensional Objects," *Computer Graphics and Image Processing,* Vol. 14, 1980, pp. 249-270.
4. C. L. Jackins and S. L. Tanimoto, "Quadtrees, Octrees, and K-trees: A Generalized Approach to Recursive Decomposition of Euclidean Space," *IEEE Trans. Pattern Analysis and Machine Intelligence,* Vol. 5, No. 5, 1983, pp. 533-539.
5. D. Meagher, "Geometric Modeling Using Octree Encoding," *Computer Graphics and Image Processing,* Vol. 19, 1982, pp. 129-147.
6. S. Tanimoto, "Image Data Structures," in *Structured Computer Vision,* S. Tanimoto and A. Klinger, eds., Academic Press, NY, 1980.
7. K. Yamaguchi, T. L. Kunii, K. Fujimura, and H. Toriya, "Octree-Related Data Structures and Algorithms," *IEEE Computer Graphics and Applications,* Vol. 4, No. 1, Jan. 1984, pp. 53-59.
8. S. N. Srihari, "Representation of Three-Dimensional Digital Images," *ACM Computing Surveys,* Vol. 13, No. 4, 1981, pp. 400-424.
9. H. Samet, "Region Representation: Quadtrees from Boundary Codes," *Comm. ACM,* Vol. 23, No. 3, 1980, pp. 163-170.
10. M. Yau and S. N. Srihari, "A Hierarchical Data Structure for Multidimensional Images," *Comm. ACM,* Vol. 26, No. 7, 1983, pp. 504-515.
11. M. Tamminen and H. Samet, "Efficient Octree Conversion by Connectivity Labeling," *Computer Graphics* (Proc. SIGGRAPH 84), Vol. 18, No. 3, July 1984, pp. 43-51.
12. R. E. Tarjan, *Data Structures and Network Algorithms,* CBMS-NSF Regional Conference Series in Applied Math, Vol. 44, SIAM, Philadelphia, PA, 1983.
13. G. H. Gonnet, *Handbook of Algorithms and Data Structures,* International Computer Science Series, Addison-Wesley, Reading, MA, 1984.

Wm. Randolph Franklin is an associate professor with the Electrical, Computer, and Systems Engineering Department at Rensselaer Polytechnic Institute. For the 1985-86 academic year he is on sabbatical with the Electrical Engineering and Computer Science Department, University of California at Berkeley. His research interests include graphics, geometry, and artificial intelligence. Franklin received the BSc in computer science from the University of Toronto, and the AM and PhD in applied mathematics (computer science) from Harvard University. He is a member of IEEE and ACM.

Franklin's address until May 1986 is Computer Science Division, Electrical Engineering and Computer Science Department, 543 Evans Hall, University of California, Berkeley, CA 94720.

Varol Akman is an associate researcher in the Department of Computer Science at the University of Utrecht. His current research is focused on algorithms and concrete complexity, especially computational geometry. Akman received the BS in electrical engineering (1979) and the MS in computer engineering (1980) from the Middle East Technical University, Ankara, Turkey. From 1980 to 1985 he was a Fulbright scholar and a research assistant at Rensselaer Polytechnic Institute, where he received the PhD in computer and systems engineering. He is a member of ACM, IEEE, and SIAM.

Akman's address is Department of Computer Science, University of Utrecht, Budapestlaan 6, PO Box 80.012, 3508 TA Utrecht, The Netherlands.