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 Parallelepips**

Wm. Randolph Franklin and Varol Akman
Rensselaer Polytechnic Institute

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. The two-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 article we demonstrate the usefulness of parallelepiped approximation in a different context: namely, solid modeling via octrees.

Octrees are data structures for modeling solids by symmetric recursive indexing. 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 $u^6$ cubes of a unit size called voxels. To obtain the octree $\Omega$, $W$ 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 $\Omega$ 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 quad-trees.

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 quad-trees. 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. 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.

**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, $j, j + 1, \ldots, k$. A list $q$ is a sequence of elements, $[x_1, x_2, \ldots, x_n]$. Element $x_i$ 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 $i$th 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 $q[i,j] = [x_i, x_{i+1}, \ldots, x_j]$.
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*.

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, $<x_1, x_2, \ldots, x_n>$, denotes $n$ elements in that order. In general, the notation of this article closely follows that of Tarjan.

We start with a description of our input and output data structures, $\pi$ and $\Omega$, respectively. It is assumed that $\pi = 2^{LMAX}$ and $g = 2^k$ where $k \leq 0 \cdots LMAX$. The elements of $\pi$ are 4-tuples called PPs:

$$\pi = \{<x, y, z, \tau> | x, y, z_1, \tau \in [0 \cdots (u-1), z_1 \leq z_2, \text{ and } x, y, z_1, \tau + 1 \text{ are all divisible by } g\}$$

The elements of $\pi$ will also be denoted by $p_i, i = 1, |\pi|$. The $x, y, z$, and $\tau$ values of a particular $p \in \pi$ will be denoted by $p(x), p(y), p(z_1)$, and $p(\tau_1)$, respectively. It is assumed that all PPs in $\pi$ are mutually disjoint.

We refer the reader to Tarjan 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 $\Omega$ of our algorithm is an octree (a tree in which every nonleaf node has eight sons) with the following properties:

1. The nodes of $\Omega$ are labeled with three types: empty, full, and partial.
2. The root of $\Omega$ is always partial except when $\pi$ is trivially equal to a completely full (respectively completely empty) $W$, in which case it becomes full (respectively empty).
3. The level of $\Omega$ is $LMAX' = \log u - \log g = LMAX - K$. (In this article we always denote $\log_2$.)
4. The leaves of $\Omega$ are either empty or full.
5. The nonleaf nodes of $\Omega$ 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 $<x, y, z>$ where $x$ is divisible by $h = 2^{LMAX-i}$; this is shorthand for PP $<x, y, z, \tau>$ where $\tau = x + h - 1$. It is noted that the $z$-length of a row at level $i$ is always $h$ units or $h/g$ spacings. Two rows $r_1 = <x_1, y_1, z_1>$ and $r_2 = <x_2, y_2, z_2>$ 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.) $2^i (i \in [1 \cdots LMAX])$ rows at level $LMAX - i$ are combinable if, when sorted in $y$ to be $r_1, r_2, \ldots$, every intermediate $r_i$ in this sequence is adjacent to its predecessor and successor and $r_i$ is a multiple of $h$.

For example, the rows $<0, 0, 0>, <0, 1, 0>, <0, 2, 0>$, and $<0, 3, 0>$ at level $LMAX - 1$ are combinable, while

October 1985
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 \(2^\text{LMAX}\) by \(2^\text{LMAX}\) by 1 voxels is obtained by combining them into a single 3-tuple \(<x, y, z>\) where \(s(y) = \min_i (r_i (y))\) and \(s(x) = r_i(x)\), and \(s(z) = r_i(z)\). Two squares \(s_1 = <x_1, y_1, z_1>\) and \(s_2 = <x_2, y_2, z_2>\) at the same level are called adjacent if \(y_1 = y_2\) and \(|x_1 - x_2| = g\). (They have the same z-length.) \(2^i (i \in [1 \ldots \text{LMAX}])\) squares are combinable if when sorted in \(x\) to be \(x_1, x_2, \ldots\) every \(s_j\) in this sequence is adjacent to its predecessor and successor, and \(s_j\) 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 \(2^\text{LMAX}\) by \(2^\text{LMAX}\) by \(2^\text{LMAX}\) voxels is obtained from their combination as a 3-tuple \(<x, y, z>\) where \(c(x) = \min_i (r_i (x))\), and \(c(y) = r_i(y)\), and \(c(z) = r_i(z)\).

If a row \(<x, y, z>\) at level \(i\), \(i \leq \text{LMAX}\), 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 \(h = 2^{\text{LMAX} - i}\). 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 \(<1, 1, \ldots 93>\) 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 \(\delta\)-lists (dimension-level lists). Here, DMAX is the maximum dimension of \(W\), and LMAX' = LMAX - \(K\), as before. A \(\delta\)-list at dimension \(D\) and level \(L\) is denoted as \(r_{D,L}\). There are LMAX' one-dimensional \(\delta\)-lists, LMAX' two-dimensional \(\delta\)-lists, and LMAX' \(-1\) three-dimensional \(\delta\)-lists when \(D = 3\). (In general, the number of the highest
The elements of \( t_{DL} \) 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: \( t_{DL} \) for read, and both \( t_{DL}^{+} \) and \( t_{DL}^{-} \) 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 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 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 \( LMAX = \frac{LMAX}{10} \log g \).

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 \( \Omega \), 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 \( \Omega \). Finally, all the cubes that were produced are added to \( \Omega \). We will show in the next section that this builds \( \Omega \) 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 \( \Omega \)” 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.

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 \( t_{LMAX} \) and \( t_{LMAX}^{-} \) cannot be combined and hence are full nodes of \( \Omega \).

**Proof:** Trivial.

**Lemma 3:** There is no need for \( t_{LMAX}^{-} \).

**Proof:** Any input to \( t_{LMAX} \) can come only from \( t_{LMAX}^{-} \), which is \([\] \) at that point. Additionally, the latter cannot send the former anything, since it cannot combine because of Lemma 2.

**Lemma 4:** \( \Omega \) is always in reduced form after STACK is applied.

**Proof:** Assume that this is not true. Take any partial node of \( \Omega \) 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 \( 2(2^{LMAX-1}) \) maximal components. Therefore, MAXCOM initializes all the one-dimensional \( \delta \lambda \)-lists with rows in \( O(LMAX|\pi|) \) operations under the assumption that append takes \( O(1) \) time.

Sorting a \( \delta \lambda \)-list is a common operation in STACK. The important point is that for \( D > 1 \), lists \( t_{DL} \) 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 \( \pi \) 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 δξ-lists. To exploit this last fact, one can use Shell sort, which is of average-case $O(n^{1.25})$. A quick sort routine that handles partly sorted lists efficiently would also be excellent.

Finally, we emphasize that after the sorting step, CSROW and CSSOR execute very efficiently, since they make a single pass over the list and spend $\sum_{\alpha \xi}$ 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,
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 Sridhar[11] and by Tamminen and Samet[12].

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

$$n_1 = n_{i_1}(x) \cdot i + n_{j_1}(y) \cdot j + n_{k_1}(z) \cdot k;$$

and

$$n_2 = n_{i_2}(x) \cdot i + n_{j_2}(y) \cdot j + n_{k_2}(z) \cdot k.$$  (Here, $i$, $j$, and $k$ are the unit vectors in $x$, $y$, and $z$ directions, respectively.)

In this case to create $\Omega$ from $\pi$, 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.

October 1985

63
assuming that \( p = <x, y, z, n_1, n_2, n_3> \), \( z_2 - z_1 > g - 1 \). (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 \( \Omega \) directly, since these must not be combined. Then for \( n' \) (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 \( n \) to obtain \( n' \).

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

3. C. L. Jackins and S. L. Tanimoto, "Octrees and Their Use in Representing Three-Dimensional Objects,"

Franklin received the BS 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.

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.

IEEE CG&A