OCTREE DATA STRUCTURES AND CREATION BY STACKING

W.R. Franklin and V. Akman

RUU-CS-86-3
March 1986

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

OCTREE DATA STRUCTURES AND CREATION BY STACKING

W.R. Franklin and V. Akman

Technical Report RUU-CS-86-3
March 1986

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
the Netherlands

Octree Data Structures and Creation by Stacking †

Wm. Randolph Franklin * and Varol Akman **

Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, Troy NY 12180 USA

* Until June 1986 visiting at: Computer Science Division, Electrical Engineering and Computer Science Dept., 543 Evans Hall, University of California, Berkeley CA 94720 USA.

** Current address: Dept. of Computer Science, University of Utrecht, Budapestlaan 6, P.O.Box 80.012, 3508 TA Utrecht, the Netherlands

INTRODUCTION

Efficient, compact data structures are necessary for the representation of octrees. First, several concrete data structures for the octree abstract data type will be compared in terms of storage space required and execution time needed to perform operations such as to find a certain node or obel. We compare information theoretic minimal representations, digital search trees sometimes storing some information in an immediate mode without pointers, and storing the set of rays, which is often the most compact.

It is also necessary to convert from other formats into octrees. Therefore, a fast method to convert a quadric surface into rays is described next. Finally, a new algorithm to stack these rays into an octree is given. It is very fast, has no intermediate swell in the storage, and does not thrash virtual memory. These algorithms have been implemented and tested on large examples. For example, the stacking program, implemented in Fortran on a Prime 750, converted an object with 12,985 rays into an octree with 106,833 obels in 3 minutes.

HISTORY

The *Octree* data structure is a space partitioning method of representing 3-D objects with the following properties:

- It is suited for representing irregular objects to a medium precision, such as one part in 256.
- Operations such as boolean combinations, transformations, and display can be implemented much more easily than with a boundary representation.
- Since the storage tends to rise with the square of the resolution, octrees are not suitable very precise objects such as machine parts.
- An octree approximates the volume of an object; the surface is not stored. Thus realistic display usually requires also storing surface normals, which can multiply the storage needed several times.
- Octrees use *spatial coherence* to reduce space and time. They are more suitable for medical objects such as human organs than for long thin objects such as geological strata.

The octree is a form of the quadtree of Bentley (1975) and Finkel, Bentley, and Stanat (1974). For some examples of quadtree algorithms, see Hunter and Steiglitz (1979) , and Samet (1980). Tanimoto (1977) applied the concept to images. The extension to representing irregular three dimensional graphic objects and implementation was by Meagher (1980, 1982a, 1982b, 1984). He formulated the basic algorithms and implemented them in a massive effort involving about 20,000 lines of code that resulted in an interactive system that allows the user to create quadtrees interactively on a graphic terminal, extend them to octrees, and then transform and combine them. Some other valuable references are Doctor and Torborg (1981) , Jackins and Tanimoto (1980) , and Yamaguchi,

† This material is based upon work supported by the National Science Foundation under grant number ECS-8351942, and by the Schlumberger-Doll Research Labs, Ridgefield, CT. The second author was also supported in part by a Fulbright award.

Kunii, Fujimura, and Toriya. (1984) Samet (1984) has an exhaustive survey of quadtrees and similar data structures.

The octree is one of several methods for representing three dimensional objects. For a survey of the methods used in some Computer Aided Design systems, see Baer, Eastman, and Henrion (1979) and Requicha (1980). Tamminen (1981, 1982) studied the related EXCELL (extendible cell method). Franklin (1980, 1981, 1983) has devised and implemented the efficient adaptive grid data structure. Hoskins (1979) describes box geometries as applied to architecture. F. Yao (1983) has presented the octant tree, which uses three planes which do not necessarily pass through its middle to divide the universe.

DEFINITIONS

An octree is logically the abstract data type called the *Digital Search Tree*. The object is contained in a universe which is a cube of size 1x1x1 with coordinates ranging from 0 to 1. The primitive component of the octree is an *Obel* of level L is a cube of linear size 2^{-L} whose lower left front corner has each coordinate of the form

$$k \cdot 2^{-L}, \quad 0 \leq k < 2^{L}$$

Every obel in the system is either *Empty, Full,* or *Partial*. When we wish to emphasize the data structure instead of the geometry, we will use *Node* for obel, and *Tree* for octree.

The universe is considered to be a level zero obel. If the object is not empty but also does not fill the universe, then the universe obel is marked as partially full. The object is now defined by the following recursive process. If L is less than the maximum level allowable, then a partial obel of level L, is divided evenly into 8 obels of level $L+1$. Each one of these obels is marked full, empty, or partial, and the process repeats on the partial ones. Partial obels at the highest level are arbitrarily declared to be full.

Each full obel is either a *Surface* obel of the object, that is it is adjacent in an orthogonal direction (not diagonally) to either an empty obel or the exterior of the universe, or else is an *Interior* obel of the object. This is not to be confused with an internal node (i.e. not a leaf) of a tree. A graphical representation of the subclassifications is shown in figure 1.

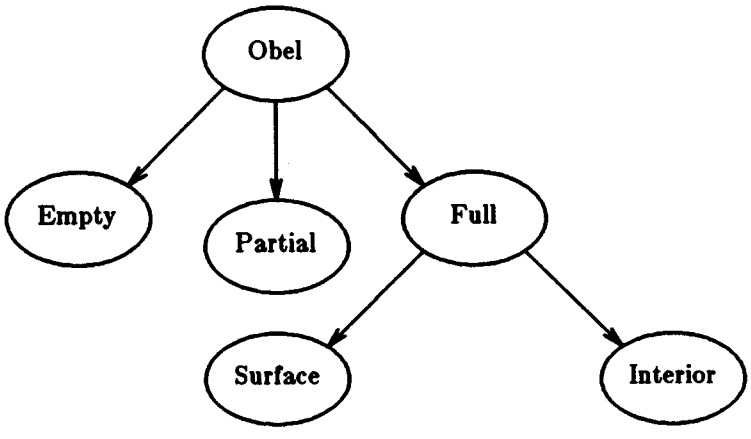

Figure 1: Types of Octree Obels

Surface obels contain associated information that is used to display the object realistically so that the user can determine its shape, especially if it is irregular. The associated information can include the local surface normal and color, which are necessary to create an understandable image.

TYPICAL NUMBERS

Some idea of the typical sizes of various components of an octree is necessary when designing an optimal implementation. Although this is dependent on the object's exact position and orientation, we can still determine some rough figures. We will let our sample object be a cube which is aligned as badly as possible: if we intend to approximate it down to level $L \geq 2$, then the cube will be the whole universe except for a single layer of level L obels removed from just inside the perimeter of the universe. Thus its side length will be $1-2^{1-L}$. This will give the number of empty obels, all at level L, as

$$N_e = (2^L)^3 - (2^L-2)^3 = 2 \cdot 4^L - 4 \cdot 2^L + 8$$

The number of full obels at level l, $2 \leq l \leq L$, is $N_{fl} = (2^l-2)^3 - (2^l-4)^3$

The total number of full obels is $N_f = \sum_{l=2}^{L} N_{fl} = 8 \cdot 4^L - 72 \cdot 2^L + 56L + 56$

The number of partial obels in the tree can be determined by working backward from L to 1. There are no level L partial obels since all the obels at the finest level are either full or empty. The level $l=L-1$ partial obels occur where the level L full and empty obels are adjacent. This occurs in a layer around the outside of the universe, and thus the level $L-1$ partial obels are precisely those obels which would be empty if the bottom level of the tree were $L-1$ instead of L. This process continues until $l=1$. Finally, at $l=0$, the whole universe is a partial obel. Thus the total number of partial obels is:

$$\begin{aligned} N_p &= 1 + \sum_{l=1}^{L-1} (2^l)^3 - (2^l-2)^3 \\ &= 2 \cdot 4^L - 12 \cdot 2^L + 8L + 9 , \quad \text{for } L>0 \end{aligned}$$

The total number of obels will be

$$\begin{aligned} N &= N_e + N_f + N_p \\ &= 12 \cdot 4^L - 56 \cdot 2^L + 64L + 73, \quad L>1 \end{aligned} \tag{1}$$

SPACE REQUIRED

In this section, we will analyze the minimal space required to store an octree, that is the information content. Let N be the number of nodes in a given octree. Since every internal node of the octree has exactly eight sons, and every exterior node (leaf), has exactly zero sons, regardless of how well balanced the tree is, then I, the number of internal nodes, is exactly $(N-1)/8$. This may be proved inductively. Therefore most of the octree is composed of leaves (that is, full or empty obels) so it is important to store them compactly.

Now the minimal storage required for a tree (apart from the nodes' attributes) is to note whether each node is full, partial, or empty, which can be done in two bits per node. If we measure storage in bytes, and assume eight bits per byte, then the simple storage required is $S_1 = 1/4N$.

Given that P_p, the probability of a partial obel (interior node) is about 1/8, the worst case for the full and empty obels is to have equal probabilities: $P_f = P_e = 7/16$. Now, using a Huffman coding, Abramson (1963) , the minimum storage per node reduces to $S_2 = 0.177N$. However, this improvement would be at the cost of a complicated coding algorithm that would be harder to implement, and the resulting octree would be harder to modify.

The associated information, such as surface normals, required for each surface node also makes this compact encoding irrelevant. A convenient representation for one normal is one byte per component, for a total of three bytes. We could precalculate the intensity at each node from some given light source and store that instead. However, storing red, green, and blue components would still require three bytes. This would also prevent us from changing the light source after the octree had been created. Therefore, S_3, the extra storage required per full surface node is 3.

Now using the P_f, P_p, and P_e given before, and assuming that almost all of the full obels are surface obels, we get

S_4 = extra storage required for surface obel attributes
= (extra storage per surface obel)
· (number of surface obels per full obel)
· (number of full obels per leaf node)
· (number of leaf nodes per general node)
· (number of nodes)
$= 3 \cdot 1 \cdot 1/2 \cdot 7/8 \cdot N$
$= 21/16\ N$

Since this totally dominates both S_1 and S_2, the simple encoding of S_1 suffices. Thus

S_5 = storage required to represent an octree with attributes
$= S_1 + S_4$
$= 25/16N$

We have not added a bit to tell whether a full node is a surface or interior node, since it can be fitted in easily.

We will now consider some possible realizations of the abstract octree.

STORING A MINIMAL OCTREE

It is possible to store an octree using only S_5 bytes by traversing the tree in a breadth first manner and packing the nodes together. The root is listed first, followed by the eight level 2 nodes, then by all the level 3 nodes in order, and so on. Unless the level 2 nodes are all partial, there will be less than 64 level 3 nodes. Thus finding a particular level 3 node requires examining all the level 2 nodes, and so on. Thus, in general, accessing a given leaf whose coordinates we know requires reading the whole tree up to that point. Thus an $\theta(logN)$ time process degenerates to a much worse $\theta(N)$ process. It is also impossible to modify the tree, for example to change a full node to a partial node with eight sons without rewriting the whole tree after that point.

While this method may be suitable for archival storage where space is critical, it is totally unsuited for computation. Thus we need another data structure for manipulation.

STORING A GENERAL DIGITAL SEARCH TREE

Instead of storing the octree as compactly as given above, we might store it as a general digital search tree, Knuth (1973). In this format, the nodes are separately allocated from a storage heap and are referenced by their address. A two byte address will be too small (there may be over 64K nodes) so we will allocate four bytes. Each partial node will contain an array of the addresses of its eight sons. All the empty nodes can be represented by one node at a distinguished address (such as 00000), as can all the interior full nodes by address 00001. Thus each partial node will require 32 bytes. Each surface full node requires at least three bytes for the normals. If we allocate four bytes, then it will be the same size as the partial nodes.

We need to distinguish between a full node and a partial node; there are two methods. First, we might allocate the two types of nodes from separate address spaces. Second, we might reserve a tag bit to identify them. The latter method is more flexible, but requires that extra bit. Since we are not using the full range of our node address space (32 bits), we can reserve the most significant bit as a tag, so these two methods are equivalent in this case.

This method requires the following storage. Since a partial node requires 32 bytes, an empty node zero bytes, an interior full node zero bytes, and a surface full node 4 bytes, then using the aforementioned probabilities with the additional worst case assumption that essentially all of the full nodes are surface, then we get:

S_6 = storage to represent the octree as a general digital search tree with attributes

$$= (\tfrac{1}{8}\cdot 32 + \tfrac{7}{16}\cdot 0 + 0\cdot 0 + \tfrac{7}{16}\cdot 4)\, N$$

$$= \frac{23}{4} N$$

that is, about 6 bytes per node.

STORING A MODIFIED DIGITAL SEARCH TREE

We observe that in the general digital search tree method given above, since there may be so many nodes, and each node is rather small, the pointer to a surface full leaf is as big as the leaf itself (4 bytes in either case). This suggests that we store leaves in *Immediate* mode, as an immediate operand is stored in an instruction in assembly language. In this format, a partial node is an array of eight 4 byte fields, one per son. For a partial son, the field is the son's address. For a full surface son, the field is the 4 byte description of the surface of the octree at that point, i.e. the surface normal. As before, empty nodes and full interior nodes are represented by distinguished values.

The storage required by this method is the same as before, except that a surface full node requires no storage since its associated information (i.e. the surface normal) is encoded in place of its address. Therefore,

S_7 = storage for modified digital search tree format with attributes

$$= 4N$$

that is, about 30% less than before. With this method, we must reserve one bit of each field to indicate whether it is an address or an immediate field. It is convenient to represent the amount of storage in terms of the maximum level number in addition to the number of nodes. Using equation (1) with the number of nodes down to level L in a generally aligned cube, we get

$$S_7 = 48\cdot 4^L - 224\cdot 2^L + 256L + 292$$

STORING A HYBRID TREE

In a general tree such as described above, most of the storage is consumed by nodes that are near the leaves, while most of the search time is spent reading nodes nearer the root. This allows us to graft the compact format onto the general digital search tree format. The process consists of using the general format, where each node has a four byte pointer to each of its sons, for the first several levels of the tree. Every leaf of this tree then becomes a separate compact subtree where all the nodes are packed together with only two flag bits per node. Although the general part of the tree will have most of the depth, it will have only a few of the nodes, and so will occupy little space. In contrast, since most of the nodes are stored in compact subtrees, they will require only 2 bits of flags instead of a 4 byte pointer. In addition, since each compact subtree is small, adding or deleting a node, which requires copying the whole compact subtree that the node occurs in, will take little time. This method is part of the database folklore, Willard (1984) , but not is easily available in a reference. We can choose the time/space tradeoff as desired by choosing the proportion of the tree to be in each format. Thus,

S_8 = storage for hybrid format, w/o attributes

$$= 1/4N$$

Adding in the storage for the attributes, we get

S_9 = storage for hybrid format with attributes

$$= S_8 + S_4$$

$$= \frac{25}{16} N$$

$$= \frac{75}{4}\cdot 4^L - \frac{175}{2}\cdot 2^L + 100L + \frac{1825}{16}$$

However, this method is much more complicated.

STORING THE OCTREE AS A SET OF RAYS

The usual method of generating an octree from an irregular object, such as a conic section is by intersecting a regular grid of rays with the object. This set of rays can then be transformed to an octree; however the set is a useful and efficient data structure in its own right. Each element has the following format:

$$(X, Y, Z_1, Z_2, N_1, N_2)$$

This represents a ray with fixed (X,Y) that enters the object at $Z=Z_1$ and leaves at $Z=Z_2$. The surface normals at the two points are N_1 and N_2, respectively. If we wish to construct an octree whose highest level is L, then

$$X,Y = k\cdot 2^{-L}\ ,\ \ k=0,1,\cdots 2^L-1$$

Z_1 and Z_2 are also rounded to the nearest multiple of 2^{-L}. Each ray requires 10 bytes composed of three bytes per normal and one byte per coordinate (assuming that $L\leq 8$). There will be about 4^L rays running through the object. This is affected by some rays not intersecting the object at all and some intersecting more than twice. Thus approximately

S_{10} = storage for set of rays format with attributes

$$= 10\cdot 4^L$$

For large L, S_{10} is half as large as S_9, the best method using the explicit octree data structure. In addition, all of the usual octree operations except rotation can be performed efficiently on the ray format. A Boolean operation on two octrees reduces to Boolean operations on pairs of rays which is trivial since the rays are one dimensional. Translation of a ray is also easy. Finally, to display an octree composed of rays, we can paint the rays into the frame buffer back to front. This works for any display angle. Another advantage of this data format is that the smaller number of data objects may allow shorter addresses, two byte instead of four byte, to be used to refer to them.

Finally, since the rays do form a somewhat regular grid, we might omit the (X,Y), and just store the (Z_1,Z_2,N_1,N_2). This reduces the total storage for the octree to about

$$S_{11} = 8\cdot 4^L$$

The special cases of objects folding back on themselves and objects that don't intersect all the rays, might be handled with exception tables if their number is limited. This is reasonably only for fairly regular objects.

RAY TRACING QUADRIC SOLIDS

An efficient algorithm for converting a quadric solid, such as an ellipsoid, was designed and implemented in Fortran-77 on a Prime 750 midicomputer by the authors. A quadric is defined by a matrix M, such that

$$(x\ y\ z\ 1)\ M \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0, \quad M = \begin{bmatrix} a & b & c & d \\ b & e & f & g \\ c & f & h & i \\ d & g & i & j \end{bmatrix}$$

After the solid is clipped to a $[0,1]^3$ universe, the output is a set of rays and surface normals as described in the last section. We assumed that each ray goes through the lower left corner of the obel; the alternative assumption would have been that the rays go through the center.

The algorithm is as follows:

a) Iterate up the quadric solid in Y. For each Y, find the 2-D conic in X and Z.

b) For each conic in the (X,Z) plane, find the range of X for which Z is real. This range contains 0, 1, or 2 segments that may be finite or infinite.

c) Iterate in X and solve the quadratic equation for Z_1 and Z_2. Clip them to $[0,1]$.

d) Calculate the normals the usual way. If Z_i was clipped, the the associated normal is $(0,0,\pm 1)$.

The above method is fast because it does not work with rays that do not intersect the solid. These are typically

the majority of all the rays. To determine the range of X for which the 2-D conic $f(X,Z)$ is real, six cases must be considered depending on the discriminant, Q, of the formal solution for Z in terms of X. That is, if

$$f(X,Z) = a\,z^2 + (b_0+b_1x)z + (c_0+c_1x+c_2x^2) = 0\,,$$

then

$$Q(X) = (b_0+b_1x)^2 - 4a(c_0+c_1x+c_2x^2)$$

The six cases and an example of each are:

a) $Q(X)$ is always positive (X^2+1).

b) $Q(X)$ is positive outside a finite interval (X^2-1).

c) $Q(X)$ is positive inside a finite interval $(-X^2+1)$.

d) $Q(X)$ is never positive $(-X^2-1)$.

e) $Q(X)$ is positive above a certain value (X).

f) $Q(X)$ is positive below a certain value $(-X)$.

For ray tracing higher order solids, such as bicubic patches, we propose to use the following observation. The functions $Z(X,Y)$ and $N(X,Y)$, for intersections and normals, although complicated, are smooth. Therefore efficient simple approximations using splines can be found. It appears that 17 evaluations of either function along a scan line, followed by linear interpolation, would allow the determination of the function at every pixel to 8 bit accuracy (instead of say 512 evaluations).

STACKING

We invented and implemented a new algorithm called *Stack* for converting the set of rays into an octree. The algorithm has the following advantages over previous algorithms:

- It is simple to program and easy to understand.
- It does not lead to intermediate storage swell.
- It accesses memory in an orderly fashion that does not thrash virtual memory when processing large objects.
- It produces a minimal octree.

Only a brief description is presented here; for more formal details and the complete code in a high level language see Franklin and Akman (1985). Another similar symmetric recursive indexing method is given in Srihari (1981). Yau and Srihari (1983) give an algorithm for constructing a tree of a d-dimensional image from trees of its (d-1) dimensional cross sections.

In this section we will consider the universe to be of size $[0,2^L]^3$ instead of $[0,1]^3$. The algorithm proceeds by iterating in dimensions as follows.

a) Each ray is partitioned into a set of *rows* which comprise a 1-D octree. That is, each row has a length which is a power of two, and a starting Z which is a multiple of its length. Rows are to obels as 1-D is to 3-D.

b) The rows are sorted by their starting point (X,Y,Z).

c) Adjacent rows are combined into squares whenever possible. A square has a side of 2^l for some $0 \le l \le L$ and starting X and Z multiples of 2^l. 2^l rows with positions

$$(i\cdot 2^l+m,\ j\cdot 2^l,\ k\cdot 2^l), \quad m = 0, \cdots, 2^l$$

can be combined into one square. The process starts at $l=L$ and interates down to $l=0$. At any l, after all the rows that can be are combined into squares, the remaining rows are each split into two rows of length 2^{l-1}. These smaller rows may be later combined into smaller squares. At the end, the remaining rows are of size one and can be considered to be obels of size one.

d) Next the squares are either combined into obels or split into squares of size one that are obels of size one.

e) Finally, the obels are formed into the new octree.

Note that the combination requires inspecting only adjacent items in memory and when a new object is created, it is appended sequentially to a list in memory. Since efficient external sorts are known, the whole process will execute efficiently in a virtual memory environment.

We implemented Stack in Ratfor. Building a 1/8 sphere with $L=6$ read 833 rays and created an octree with 6569 obels. The execution time was 9.2 seconds on a Prime 750. A higher resolution sphere consisting of 12,985 rays was transformed into an octree with 106,833 obels in 3 minutes. This octree has $N_f=67{,}570$, $N_p=13{,}354$, and $N_e=25{,}909$ and is larger than many of the examples cited in Tamminen (1984) and Yau (1983).

THE SYSTEM

These features have been integrated into a system with the following data flow as shown in figure 2.

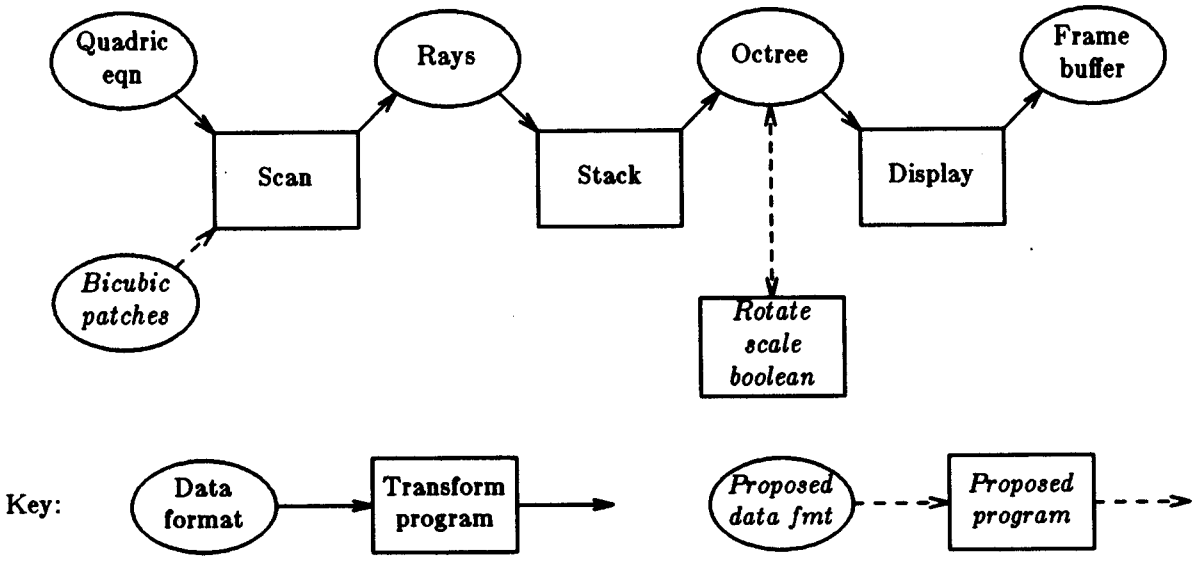

Figure 2: Dataflow in the System

The display algorithm is a back to front paint into a frame buffer. Depending on the octant in which the viewer is situated, there is a certain permutation of the numbers from 1 to 8 that is easy to determine. If the octree is traversed recursively with the eight sons of each internal node being visited in this order, then the leaves will be visited in back to front order. Note that traversing the tree is this order is no harder than traversing the tree in the usual lexicographic order.

The alternative display algorithm is to paint the image front to back and never paint any pixel twice, Meagher (1982). A quadtree in the frame buffer records which pixels have been painted once. Our method paints some pixels twice but has one less complicated (and slow to manage) data structure.

SUMMARY

Of the various data structures considered with which to implement the abstract octree, we see that the most efficient are 1) the digital search tree with surface full nodes stored as immediate data, and 2) a set of the rays. We have also presented a very fast stacking algorithm for converting a set of rays into an octree.

REFERENCES

Abramson, N. (1963) *Information Theory and Coding,* McGraw-Hill Electronic Sciences Series.

Baer, A., Eastman, C., and Henrion, M. (Sept. 1979) "Geometric Modelling: A Survey," *Computer Aided Design,* vol. 11, no. 5.

Bentley, J.L. and Stanat, D.F. (July 1975) "Analysis of Range Searches in Quad Trees," *Information Processing Letters,* vol. 3, no. 6, pp. 170-173.

Doctor, L. J. and Torborg, J. G. (1981) "Display Techniques for Octree Encoded Objects," *IEEE Computer Graphics and Applications,* vol. 1, no. 3, pp. 29-38.

Finkel, R.A. and Bentley, J.L. (1974) "Quad Trees: A Data Structure For Retrieval On Composite Key," *Acta Informatica,* vol. 4, pp. 1-9.

Franklin, Wm. Randolph (July 1980) "A Linear Time Exact Hidden Surface Algorithm," *ACM Computer Graphics,* vol. 14, no. 3, pp. 117-123.

Franklin, Wm. Randolph (April 1981) "An Exact Hidden Sphere Algorithm That Operates In Linear Time," *Computer Graphics and Image Processing,* vol. 15, no. 4, pp. 364-379.

Franklin, Wm. Randolph and Akman, Varol (October 1985) "Building an Octree from a Set of Parallelepipeds," *IEEE Computer Graphics and Applications.*

Franklin, Wm. Randolph (16-21 October 1983) "Adaptive Grids For Geometric Operations," *Proc. Sixth International Symposium on Automated Cartography (Auto-Carto Six),* vol. 2, pp. 230-239, Ottawa, Canada.

Hoskins, E.M. (November 1979) "Design Development and Description Using 3D Box Geometries," *Computer Aided Design,* vol. 11, no. 6, pp. 329-336.

Hunter, G.M. (April 1979) "Operations on Images Using Quad Trees," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. PAMI-1, no. 2, pp. 145-153.

Jackins, C. L. and Tanimoto, S. L. (1980) "Quadtree, octree, and K-trees: A Generalized Approach to Recursive Decomposition of Euclidean Space," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 5, no. 5, pp. 533-539.

Knuth, D.E. (1973) *The Art of Computer Programming, Volume 3: Sorting and Searching,* Addison-Wesley.

Meagher, Donald J. (October 1980) *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer,* IPL-TR-80-111, Rensselaer Polytechnic Institute, Image Processing Lab.

Meagher, Donald J. (April 1982) *The Octree Encoding Method for Efficient Solid Modelling,* Rensselaer Polytechnic Institute, Electrical, Computer, and Systems Engineering Dept., Ph.D. thesis.

Meagher, Donald J. (1982b) "Geometric Modelling Using Octree Encoding," *Computer Graphics and Image Processing,* vol. 19, pp. 129-147.

Meagher, Donald J. (November 1984) "The Solids Engine: A Processor for Interactive Solid Modelling," *Proc. Nicograph.*

Requicha, Aristides A. G. (December 1980) "Representations for Rigid Solids: Theory, Methods, and Systems," *ACM Computing Surveys*, vol. 12, no. 4, pp. 437-464.

Samet, H. (December 1980) "Deletion in Two-Dimensional Quad Trees," *Comm. ACM*, vol. 23, no. 12, pp. 703-710.

Srihari, S. N. (1981) "Representation of Three-Dimensional Digital Images," *ACM Computing Surveys*, vol. 13, no. 4, pp. 400-424.

Tamminen, M. (1981) *Expected Performance of Some Cell Based File Organization Schemes,* REPORT-HTKK-TKO-B28, Helsinki University of Technology, Laboratory of Information Processing Science, SF-02150 Espool 5, Finland.

Tamminen, M. (June 1982) "The Excell Method for Efficient Geometric Access to Data," *ACM IEEE Nineteenth Design Automation Conference Proceedings*, pp. 345-351.

Tamminen, M. and Samet, H. (1984) "Efficient Octree Conversion by Connectivity Labelling," *ACM Computer Graphics*, vol. 18, no. 3, pp. 43-51. (SIGGRAPH'84 Proceedings)

Tanimoto, S.L. (June 1977) "A Pyramid Model for Binary Picture Complexity," *Proc. IEEE Computer Society Conference on Pattern Recognition and Image Processing,* Rensselaer Polytechnic Institute.

Willard, Dan E. (1984) *personal communication,* State University of New York at Albany.

Yamaguchi, K., Kunii, T. L., Fujimura, K., and Toriya, H. (1984) "Octree-related Data Structures and Algorithms," *IEEE Computer Graphics and Applications*, vol. 4, no. 1, pp. 53-59.

Yao, F. F. (April 1983) "A 3-Space Partition and Its Applications (extended abstract)," *ACM 15th Symposium on the Theory of Computing*, pp. 258-263, Boston.

Yau, M. and Srihari, S. N. (1983) "A Hierarchical Data Structure for Multidimensional Images," *Comm. ACM*, vol. 26, no. 7, pp. 504-515.