Comput. & Graphics Vol. 13, No. 3, pp. 373–379, 1989
Printed in Great Britain.

0097-8493/89 $3.00 + .00

Technical Section

REPRESENTING OBJECTS AS RAYS, OR HOW TO PILE UP AN OCTREE?

VAROL AKMAN
Department of Computer Engineering and Information Sciences, Bilkent University, P.O. Box 8, 06572 Maltepe, Ankara, Turkey

and

WM. RANDOLPH FRANKLIN
Department of Electrical, Computer, and Systems Engineering, Rensselaer Polytechnic Institute, Troy, New York 12180-3590, USA

Abstract—Quadtrees, octrees, and in general k-trees have established themselves as useful hierarchical data structures in computer graphics, image processing, and solid modeling. A fundamental operation in a system based on k-trees is the construction of a k-tree. Here, we review a new way of doing this operation. Basically, we have invented a method to store an object as a set of *rays* and an algorithm for converting such a set into a k-tree. (For example, in 3D a ray is a thin parallelepiped.) The algorithm is conceptually simple, works for any k, and piles up, using an approach we call *stacking*, a k-tree from the rays very fast. It produces a minimal k-tree and does not lead to intermediate storage swell. For large-scale realistic objects, which consist of many thousands of rays, the algorithm debunks the "expensive octree creation" myth.

1. INTRODUCTION

Quadtrees, octrees, and in general, k-trees are data structures† all based on the symmetric recursive partitioning of space. We are concerned in this paper with a task of fundamental importance in systems based on octrees. This is the operation of constructing (creating, building) an octree. We have observed, somewhat surprisingly, that octree construction has not received the importance it deserves and that the algorithms cited in recent overviews[1, 2] are neither efficient nor conceptually easy to understand and implement.

We thus aim to accomplish two things here: (i) propose a new way of storing an object as a set of *rays*‡, and (ii) describe an original algorithm to convert this set of rays into an octree. Our algorithm is so pure and simple that it is indeed astonishing that it has not been thought of before. We shall substantiate this claim to simplicity and ease of understandability with a data flow diagram (cf. Fig. 8) which succinctly shows the correctness of our algorithm.

Our previous papers on the theme of this paper are [3, 4]. Although this presentation will essentially be self-contained, we refer the interested reader to [3, 4] for topics which will not be covered here (*e.g.*, information-theoretic minimal representations of k-trees[4]).

† In order to make the upcoming presentation more concrete and the definitions less cumbersome, hereafter we shall choose octrees ($k = 3$) as the representative member of this group. It will be observed, however (and we shall explicitly note this at times), that our results will be applicable to both quadtrees ($k = 2$) and k-trees with $k > 3$ as well, *mutatis mutandis*.

‡ In a previous publication[3], we have called the rays *parallelepipeds*. It will be seen in the sequel that the latter term is indeed more suggestive but for the sake of conciseness we shall adopt the former.

2. TERMINOLOGY

We recall some definitions and cite, for the sake of completeness, well-known facts regarding octrees while probably running the risk of boring the reader; this section should be skipped unless one is unfamiliar with octrees.

An *octree* is a special case of the abstract data type *digital search tree*. (Knuth[5] gives an excellent account of digital search trees.) The object, which we shall denote as Σ, is contained in a *universe*, Υ, which is a cube of size $U \times U \times U$ where $U = 2^{\ell}$. Here ℓ is a positive integer. (With today's technology, an ℓ value of 10 or 11 is more or less standard but this doesn't have to concern us at the moment.) Υ is divided into U^3 small cubes of unit volume called *voxels*. To obtain an octree, which we shall denote as Ω, Υ is continuously subdivided into eight symmetric *octants* of equal volume. Each of these octants will be either homogeneous (*i.e.*, either fully occupied by Σ or void) or heterogeneous (*i.e.*, partly occupied by Σ). We further subdive the heterogeneous octants into suboctants. This is stopped when octants (possibly voxels) of uniform properties are obtained. Obviously, Ω is only an approximation to Σ since at the voxel level a partly full voxel should be labeled either full or empty.

Following the established usage§, let us consider the *obels*. An obel can be *empty, full,* or *partial;* cf. Fig. 1. Υ is considered to be a level-0 obel. If Σ is not void but does not fill Υ either, then the universe obel is labeled as *partial*. Then the following recursive process is carried out. If $l < \ell$ then a partial obel at level-l is divided evenly into eight obels at level $l + 1$. Each of these obels is again labeled *full, empty,* or *partial* and the process is repeated on the partial ones. Let us as-

§ Sometimes, we can't help feeling fortunate that we did not invent the words voxel and obel.

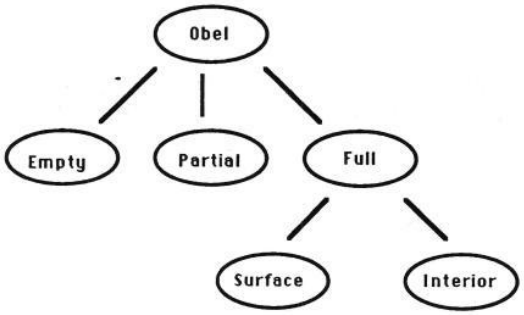

Fig. 1. Types of octree obels.

sume, without loss of generality, that the partial voxels, if any, at level-ℓ are arbitrarily declared to be full.

It is emphasized that the level of a obel v in an octree is defined recursively as $level(v) = 0$, in case v is the root and as:

$$level(v) = level(father(v)) + 1,$$

otherwise. The *depth* of an octree is understood as the level of its deepest (lowest) leaf. Thus, any octree which has voxels has depth ℓ, by definition.

The quadtree complexity theorem given in [11] states that the size of a quadtree representation of a region is linear in the perimeter of the region (in all but pathological cases). In general, it turns out that [15] the size of a k-tree of a set of k-dimensional objects is proportional to the sum of the resolution and the size of the $(k-1)$-dimensional interfaces between these objects (again, in all but pathological cases).

3. RELEVANT RESEARCH

Quadtrees, octrees, and to a lesser extent¶ k-trees have proved to be a fertile area of research with many hundreds of papers and various surveys. A recent, rather nontechnical overview is provided by Samet and Webber [1, 2] who cite about 200 references (although with some important omissions). Since we carry no pretensions of covering all the relevant work, we shall be selective and refer the reader to Requicha [7] and Srihari [8] for two informative surveys on the representation of rigid solids, *viz.*, 3D digital images. The reader should also benefit from Yamaguchi *et al.* [18] and the other overview papers of Samet [9, 10] who is, without doubt, the most prolific survey author in this area.

Early work on quadtree algorithms was done by Hunter and Steiglitz [11] who showed how to perform a set of operations on images using quadtrees. Jackins and Tanimoto [12, 13] extended this work to octrees. Doctor and Torborg [14] presented, in an elegant article, display techniques for octree-encoded objects.

Construction of a serious software system which lets irregular 3D objects to be represented and manipulated was first accomplished by Meagher [15]. He implemented a large program and obtained a graphics system that allows the user to build quadtrees interactively, extend them to octrees, and carry out essential transformations and set-theoretic operations on them. Later, Meagher built a special hardware, "The Solids Engine," which is an impressive system for interactive solid modeling based on octrees.

Construction of quadtrees or octrees has been treated in [1, 2] in some detail. To quote Samet and Webber [1, p. 64ff]:

> "The algorithm for building a raster quadtree from a 2D array can be derived directly from the definition of the raster quadtree. When building a quadtree from raster data presented in raster scan order (*i.e.*, the array is processed row by row), we use the bottom-up neighbor finding algorithm to move through the quadtree in the order in which the data is encountered. [...] Such an algorithm takes time proportional to the number of pixels in the image. Its execution time is dominated by the time to check whether nodes should be merged. This can be avoided by predictive techniques that assume the existence of a homogeneous node of maximum size whenever a pixel that can serve as an upper left corner of a node is scanned (assuming a raster scan from left to right and top to bottom). In such a case, merging is reduced and the algorithm's execution time is dominated by the number of blocks in the image rather than by the number of pixels. However, this algorithm does require an auxiliary data structure (which can be implemented by a fixed-size array) of a size on the order of the width of the image, to keep track of all active quadtree blocks (*i.e.*, blocks containing pixels that have not yet been encountered by the raster scanning process)."

Techniques for building octrees include merging the cross-sectional images (which are represented as quadtrees) of the object in sequence. This technique is examined by Yau and Srihari [16]; they show how to construct a k-dimensional tree representation from multiple $(k-1)$-dimensional cross-sectional images. It is noted that there is a rather superficial resemblance between the algorithm in [16] and our algorithm. Samet [19–21] considers the conversion of rasters, binary arrays, and boundary codes to quadtrees; however, his methods are slow. Tamminen and Samet [17] give an algorithm for converting from the boundary representation of a solid to the corresponding octree model using a technique called "connectivity labeling." Shaffer and Samet [22] consider optimal quadtree construction methods.

Sometimes, it is possible to use a small number of 2D images to reconstruct an octree representation of a 3D object. This is done by taking silhouettes (*i.e.*, projection images) from various viewpoints. These silhouettes are then processed in order to create a bounding volume that serves as an approximation of the object. Potmesil [23] gives a fine account of how this can be achieved.

4. RAY REPRESENTATION

Ray representation has, in fact, been known for a long time—the catch is that it has been used for another purpose, namely, numerical integration. For example, to compute the volume of a 3D object, Σ, one ap-

¶ For $k = 4$ an obvious application is as follows. Let the fourth dimension represent time. Then it is possible to view a changing 3D scene as a 4D object. Samet and Tamminen wrote an article [6] on this subject.

64-2

proximates it with a set of rectangular parallelepipeds[7]. These parallelepipeds (or, as noted before, rays, from now on) are assumed, without loss of generality, to be evenly spaced in the xy-plane but to have varying lengths along the z-direction. Needless to say, one postulates some conditions on the shape of Σ such as its boundary is composed of well-behaving surfaces, and so on. The reader is referred to Roth[24] for a particularly clear exposition of a related idea, *viz.*, ray casting. Fig. 2 shows the rays for a 2D object (region). In 3D these thin rectangles would become thin parallelepipeds. We assume throughout this paper that the rays are cast at unit spacing. (In [3] we treat the general case which makes our algorithm only more tedious.)

We use rays as follows. Let each ray be given in the following format:

$$\rho = (x, y, z_1, z_2, n_1, n_2),$$

where $x, y, z_1, z_2 \in [0, U-1]$. This corresponds, as pointed out in the preceding paragraph, to a ray with fixed (x, y) that enters the object at z_1 and leaves it at z_2. (Assume that $z_1 \leq z_2$.) When an object is given as a set of rays, it is assumed that all rays in the set are distinct and disjoint. The *surface normals* n_1 and n_2 are associated with z_1 and z_2, respectively. They are used to display Σ realistically so that the user can determine its shape, especially if it is an irregular object. In fact, one can even incorporate the color values in addition to the local surface normal, for each full surface obel. In the sequel, we shall ignore the surface normals since they can be easily handled (as shown in [3, pp. 63–64]) by our algorithm with no extra effort.

There are several advantages of the ray format:

- Set-theoretic (Boolean) operations on two octrees can be performed via trivial operations on rays since rays are 1D.
- Translation of a ray is easy. (Rotation is still problematic; cf. Gargantini[25] for translation and rotation of quadtrees.)
- To display an octree stored as a set of rays, we simply paint the rays into the frame buffer back to front. Notice that this would work for any display angle.†

5. RAYS FROM QUADRIC SOLIDS

The second author wrote an efficient program for converting a quadric solid (*e.g.*, an ellipsoid) to rays. This program is in Fortran-77 and runs on a Prime 750.

The reader is no doubt familiar with the definition of quadrics. Briefly, let Ξ be a 4×4 matrix and let V be the vector $[xyz1]^T$. (The superscript T denotes the transpose.) Now a concise way of writing down the equation of a quadric is:

$$V^T \Xi V = 0.$$

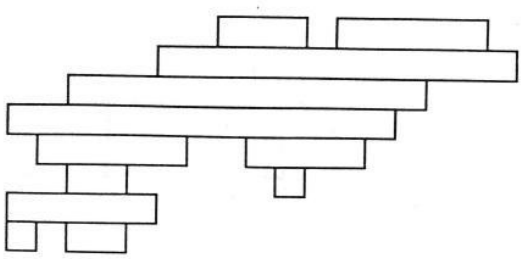

Fig. 2. Ray representation for a region (region boundary not shown).

The program mentioned above clips a solid to the unit cube (*i.e.*, $[0, 1]^3$) and outputs a set of rays and surface normals as described before. The algorithm is as follows:

0. Iterate up the quadric solid in y. For each y, find the 2D conic in x and z.
1. For each conic in the xz-plane, find the range of x for which z is real. It is noted that this range contains up to two segments that may be finite or infinite. (We do not dwell here on how to determine the range of x for which the 2D conic $f(x, y)$ is real but simply state that six cases should be considered depending on the discriminant of the formal solution for z in terms of x; cf. [4] for details.)
2. Iterate in x and solve the quadratic equation for z_1 and z_2. Clearly, they are clipped to $[0, 1]$.
3. Calculate the normals in the obvious way. If z_i were clipped in step 2, then the associated normals would be $(0, 0, \pm 1)$.

Why is the above algorithm fast? Because it doesn't work with the rays that do *not* intersect the object. For reasonable objects, these are typically the majority of all the rays.

Caveat: The reader is probably wondering how to obtain the rays for higher order solids whose boundaries are, say, bicubic patches. Our observation is that, although the functions $z(x, y)$ and $n(x, y)$ for intersections and normals, respectively, would be complicated then, they are nevertheless smooth. Therefore, efficient simple approximations using splines can be found[4].

6. COMBINING AND SPLITTING

At the core of our algorithm for piling up an octree are two operations: *combining* and *splitting*. We now explain them.

Given a ray ρ, we need the *maximal rows* of ρ. These are computed recursively as follows. First search for the longest (in z) row in ρ and remove it from ρ. This is a maximal row. Now ρ is either reduced to a shorter ray or divided into two rays both shorter than the original. In both cases, the search continues until maximal rows of z-length equal to 1 are obtained. It is noted

† Depending on the octant in which the viewer is situated, there is a certain permutation of numbers from 1 through 8 that is easy to determine. If the octree is traversed recursively with the eight children of each partial obel being visited in that permutation order, then the leaves will be visited in back to front order. An alternative display algorithm[15] is to paint the image front to back and never paint any pixel twice.

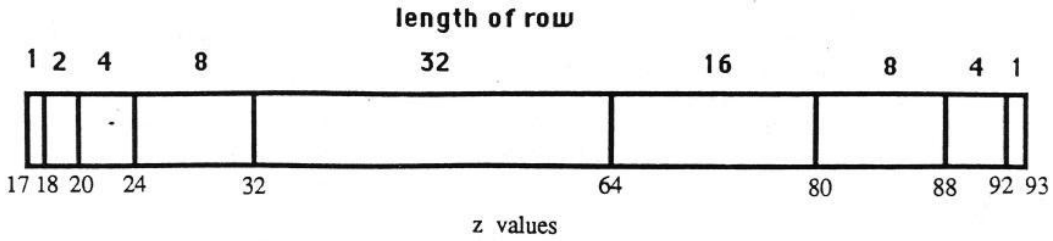

Fig. 3. Splitting a ray (1, 1, 17, 93) into nine maximal rows (not to the scale).

that, once the maximal rows of ρ are found, it should be impossible to obtain a longer row by putting two maximal rows together.

For example, Fig. 3 shows the nine maximal rows obtained from the ray (1, 1, 17, 93). They are as follows:

rows (1, 1, 17) and (1, 1, 92) ..... at level ℓ
row (1, 1, 18) ..... at level $\ell - 1$
rows (1, 1, 20) and (1, 1, 88) ..... at level $\ell - 2$
rows (1, 1, 24) and (1, 1, 80) ..... at level $\ell - 3$
row (1, 1, 64) ..... at level $\ell - 4$
row (1, 1, 32) ..... at level $\ell - 5$

A *row* at level-i is a triple (x, y, z) where z is divisible by $2^{\ell-i}$. Clearly, this is shorthand for the ray $(x, y, z, z + 2^{\ell-i} - 1)$; in other words, the z-length of a row at level-i is always $2^{\ell-i}$. Two rows, $\rho_1 = (x_1, y_1, z)$ and $\rho_2 = (x_2, y_2, z)$, at the same level are called *adjacent* if $x_1 = x_2$ and $|y_1 - y_2| = 1$. (Note that they both have the same z.) A set of 2^i rows at level $\ell - i$ can be *combined* if, when sorted by y to get a set of rows, every row in this set is adjacent to its predecessor and its successor. When rows are combined one obtains squares which are explained below.

For example, the rows (0, 0, 0), (0, 1, 0), (0, 2, 0), and (0, 3, 0) at level $\ell - 1$ can be combined, while the rows (0, 1, 0), (0, 2, 0), (0, 3, 0), and (0, 4, 0) at level $\ell - 1$ cannot (Fig. 4).

Let $\rho_1, \rho_2, \cdots$ be a set of 2^i combinable rows at level $\ell - i$. A *square*, σ, at level $\ell - i$ is obtained by combining them into a single triple (x, y, z) where $x = \rho_1$'s x, $z = \rho_1$'s z, and $y = \min_i \rho_i$'s y. Two squares, $\sigma_1 = (x_1, y_1, z)$ and $\sigma_2 = (x_2, y_2, z)$ at the same level are called *adjacent* if $y_1 = y_2$ and $|x_1 - x_2| = 1$. A set of 2^i squares at level $\ell - i$ can be *combined* if, when sorted by x to get a set of squares, every square in this set is adjacent to its predecessor and its successor. As a result of combining squares we obtain cubes which are defined below.

For example, the squares (0, 0, 0), (1, 0, 0), (2, 0, 0), and (3, 0, 0) at level $\ell - i$ are combinable while the squares (1, 0, 0), (2, 0, 0), (3, 0, 0), and (4, 0, 0) are not (Fig. 5).

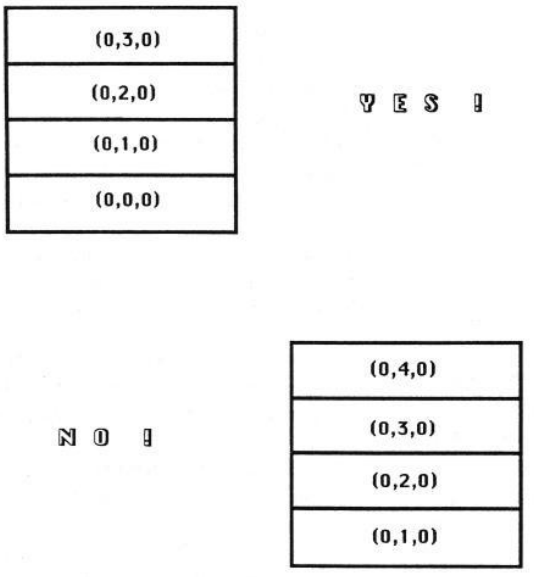

Fig. 4. Four rows of length four that are combined into one square versus four rows that cannot be combined, since they are misaligned.

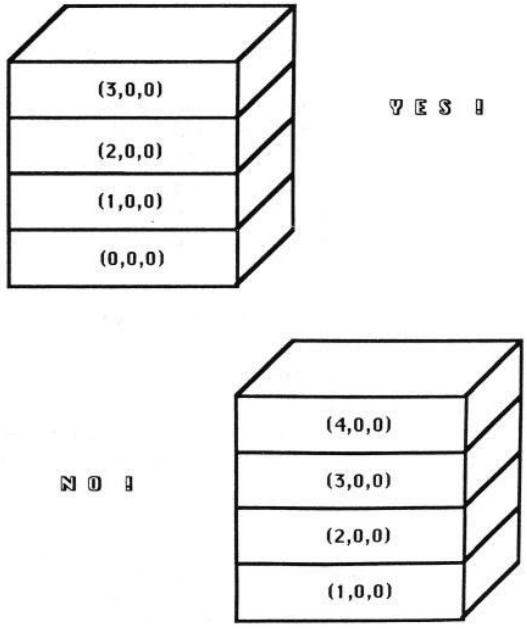

Fig. 5. Four squares of length four that are combined into one cube versus four squares that cannot be combined, since they are misaligned.

64-4

Let $\sigma_1, \sigma_2, \cdots$ be 2^i combinable squares at level $\ell - i$. A *cube*, κ, is obtained as the triple (x, y, z) where $y = \sigma_1$'s y, $z = \sigma_1$'s z, and $x = \min_i \sigma_i$'s x.

If a row (x, y, z) at level-i is *split* in the z direction, then two rows, (x, y, z) and $(x, y, z + \hbar)$, 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), $(x, y + \hbar, z)$, $(x + \hbar, y, z)$, and $(x + \hbar, y + \hbar, z)$, are obtained at level $i + 1$. Here $\hbar = 2^{\ell - i - 1}$. Obviously, the idea of splitting is generalizable to cubes and hypercubes, once we are at ≥4D. See Fig. 6 for illustrations of splitting.

The main data structure is a set of lists which will be called $\Delta\Lambda$-lists (dimension-level lists). An individual list in this set is denoted as $\iota_{D,L}$, where D denotes the dimension and L denotes the level it belongs to.

7. THE STACKING ALGORITHM

We assume that each ray has already been divided into its maximal rows (cf. the example shown in Fig. 3) and these maximal rows have already been inserted into the relevant 1D $\Delta\Lambda$-lists using the procedure **maxcom** which is fully described in [3, p. 62].

Our stacking algorithm first tries to combine adjacent rows into squares. If a row cannot be combined, it will be split into two smaller (half-size) rows which are tried until the remaining pieces are at level-ℓ. These are inserted into Ω since there is no way they can be combined.

Then the stacking algorithm tries to combine adjacent squares into cubes. Any square that cannot be thus combined will be split into four smaller (quarter-size) squares and the process will be repeated until the remaining pieces are at level-ℓ. These latter pieces are added to Ω. Clearly, the obtained cubes are also added to Ω. It is noted that the elements of $\iota_{D,L}$ are rows when $D = 1$, squares when $D = 2$, cubes when $D = 3$, and hypercubes when $D > 3$. Our algorithm is still correct when $D > 3$ as a result of its general approach. Another important point to observe is that we need $(k - 1) \times (\ell + 1)$ lists in k-dimensional space. Fig. 7 shows the usual case where $k = 3$ and $\ell = 10$.

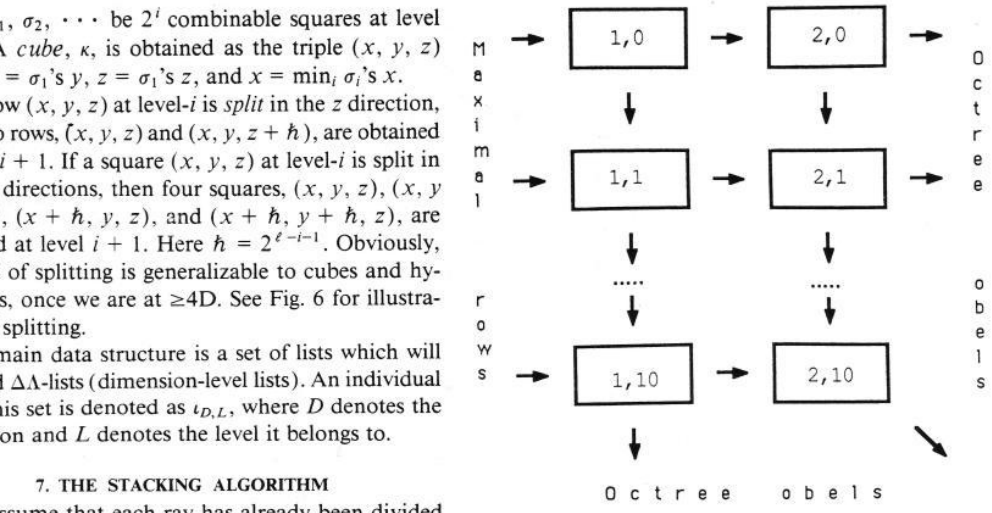

Fig. 7. A diagram showing the dimension-level lists and the flow of data for the common case of dimension = 3 and level = 10.

This process will build the octree, Ω, in its reduced‡ form.

When there are many rays, it may be suitable to use linear disk files to implement $\Delta\Lambda$-lists. Only three files will be open during the execution of the stacking algorithm: $\iota_{D,L}$ for read operations and $\iota_{D+1,L}$ and $\iota_{D,L+1}$ for write operations (cf. Fig. 8). Since the reads always take place sequentially and the writes are always carried out as appends, the algorithm is safe against virtual-memory page faults.

The algorithm works by iterating on dimensions as follows (cf. procedures **csrow** and **cssqr** in [3, p. 63] for precise descriptions of steps 2 and 4 below):

0. Each ray is partitioned, as noted above, into a set of rows which comprise a 1D octree. Thus, each row has a length which is a power of 2 and a starting z value which is a multiple of its length. In a nutshell, rows are to obels as 1D is to 3D.
1. The rows are sorted into lexicographic order by (y, z, x). This is necessary to detect combinable rows in one-pass.
2. (Combine/split process for rows): Adjacent rows are combined into squares whenever this is possible.

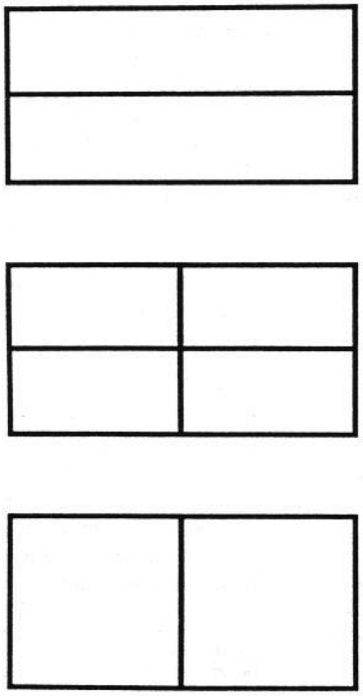

Fig. 6. Two rows of length four that cannot be combined into a square but can be split into four rows of length two and combined into two squares of side two.

‡ An octree is in its *reduced* form if it has no partial nodes with all empty or all full children.

645

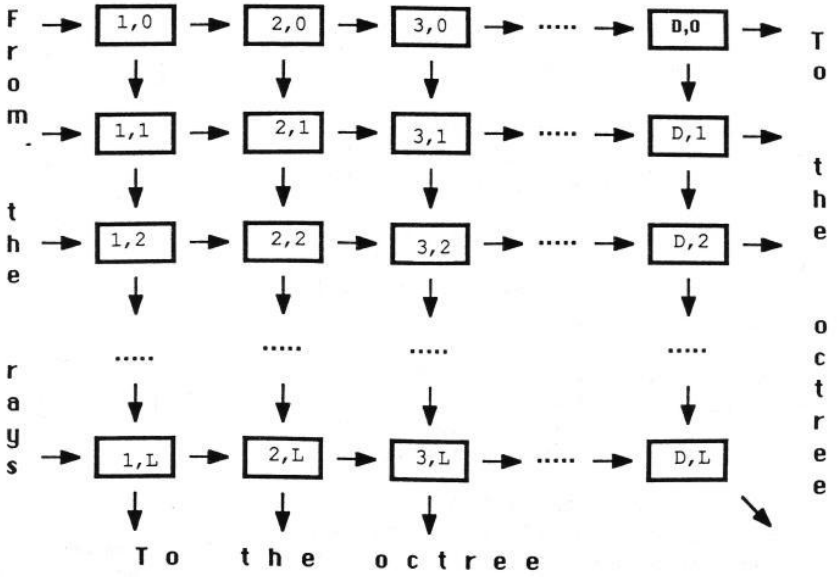

Fig. 8. A diagram showing the data flow of the octree generation algorithm (D = max dimension, L = max level).

A square has a side of 2^l for some $0 \le l \le \ell$ and starting x and z coordinates that are multiples of 2^l. If we find 2^l rows with positions

$$(i \times 2^l + m, j \times 2^l, k \times 2^l),$$

where $m = 0, \ldots, 2^l$, then we can combine them into one square. Combining rows is illustrated in Fig. 4. The process of combining/splitting starts at $l = 0$ and iterates up to ℓ. At any l, after all the rows that can be combined are indeed combined into squares, the remaining squares are each split into two rows of length 2^{l-1}. These smaller rows, it may turn out, may later be combined into smaller squares (cf. Fig. 6). At the end, the remaining rows are of size 1 and can be considered as voxels.

3. The squares are sorted into lexicographic order by (z, x, y). This is necessary to detect combinable squares in one-pass.
4. (Combine/split process for squares): Next the squares are either combined into obels or split into squares of size 1 that are voxels. Combining squares is illustrated in Fig. 5.
5. Finally, the obels are formed into the new octree. Given an obel, inserting it into an incomplete octree is a simple problem, cf. Knuth[5] for the analogous operation in digital search trees.

It is emphasized that the combining operation requires inspecting only adjacent items in memory and when a new square (or cube) is created, it is appended sequentially to a list in memory. Since efficient external sorts are known[5], the whole process executes efficiently in a virtual memory environment.

8. TIMING AND EXTENSIONS

We have presented a novel algorithm for constructing a k-tree from a set of rays approximating a k-dimensional object. Our algorithm is simple to program and easy to implement. It is very suitable for handling precisely specified objects, that is, objects consisting of many thousands of rays, since it can work with linear files which are accessed in an orderly manner.

We have implemented the stacking algorithm in Ratfor§ on a Prime 750. Building a 1/8 sphere took 9.2 seconds of CPU time; this object has 6569 obels and is made of 833 rays. For a paraboloid built from 916 rays, the final octree has 5913 obels and the timing is 7.4 seconds of CPU. In each case the I/O time is small: 0.9 and 0.3 second, respectively.

A high resolution sphere consisting of 12985 rays took about 3 minutes of CPU; the octree has 106833 obels with the following distribution: 67570 full, 25909 empty, and 13354 partial. This is larger than many of the examples cited in Yau and Srihari[16], and Tamminen and Samet[17].

Our algorithm has certain features which makes it suitable for a hardware implementation. Central among them is the heavy use of sorting; there is now much work in fast sorting machines. A hardware extension to the stacking algorithm would then be as follows. Consider a scene of several objects, $\Sigma_1, \Sigma_2, \ldots$, which is constantly updated due to say, motion. That is, objects change their locations as a result of their movement and we want to update their images (conveniently

§ Ratfor is a structured dialect of Fortran, implemented by T. Beyer and used throughout B. W. Kernighan and P. J. Plauger's *Software Tools*, Addison-Wesley, Reading, MA (1976).

64-6

rendered on a raster graphics screen). We assume that the underlying representation scheme is octrees. Now, instead of building the octrees once and then updating them, we keep the objects as rays and update the rays and run our algorithm at each time step to obtain the new octrees.

Probably the best approach to make this paradigm work would be to have a parallel processing environment in which each box in Fig. 8 is assigned to one processor. We first put Σ_1 to the "pipeline." While this object is being converted to its octree, Ω_1, the next object, Σ_2, can be introduced into the pipeline, since some processors will be finished with processing Σ_1's rays. Then we introduce Σ_3, and so on. This is somewhat reminiscent of the systolic algorithms in VLSI.

It is noted that each processor is simply a sort machine. We assume the existence of (i) a fast ray casting machine to convert objects to rays (Section 5 shows that this is not far-fetched), and (ii) a fast rendering machine which, given an octree, paints it on the screen. We are aware of the sweeping remarks we have made in describing this extension but we trust that it may be feasible. Details remain to be filled. . . .

For example, in Fig. 8, as soon as $\iota_{1,0}$ is finished with its computation (*i.e.*, as soon as it forwards the combined (resp. split) rows to $\iota_{2,0}$ (resp. $\iota_{1,1}$)) it can receive the maximal rows for another object. This clearly generalizes to the other processors. The result is a computation "wave" which starts at $\iota_{1,0}$ and moves in rows perpendicular to the northwest-to-southeast diagonal of Fig. 8.

9. CONCLUSION

Of the various data structures to implement the abstract k-tree, a set of rays (along with an algorithm for converting this to a k-tree) seems to be the most efficient. "While such data structures are not necessary for the processing of simple scenes, they are central to the efficient processing of large-scale realistic scenes"[1, p. 48].

Acknowledgements—The first author is grateful to Özay Oral and Mehmet Baray for their unfailing support. The second author's research is supported by the National Science Foundation under PYI grant number DMC-8351942. It should be remarked that the mention of commercial products in this paper does not necessarily imply endorsement.

REFERENCES

1. H. Samet and R. E. Webber, Hierarchical data structures and algorithms for computer graphics—Part I: Fundamentals. *IEEE Comp. Graphics and Appl.* **8**(3), 48–68 (May 1988).
2. H. Samet and R. E. Webber, Hierarchical data structures and algorithms for computer graphics—Part II: Applications. *IEEE Comp. Graphics and Appl.* **8**(4), 59–75 (July 1988).
3. Wm. R. Franklin and V. Akman, Building an octree from a set of parallelepipeds. *IEEE Comp. Graphics and Appl.* **5**(10), 58–64 (October 1985).
4. Wm. R. Franklin and V. Akman, Octree data structures and creation by stacking, in *Computer-Generated Images: The State of the Art*, N. Magenat-Thalmann and D. Thalmann (Eds.), Springer-Verlag, Tokyo, 176–185 (1985).
5. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA (1973).
6. H. Samet and M. Tamminen, Bintrees, CSG trees, and time. *ACM Comp. Graphics* (*Proceedings of SIGGRAPH'85*) **19**(3), 121–130 (July 1985).
7. A. A. G. Requicha, Representation of rigid solids: Theory, methods, and systems. *ACM Comp. Surveys* **12**(4), 437–464 (December 1980).
8. S. N. Srihari, Representation of three-dimensional digital images. *ACM Comp. Surveys* **13**(4), 400–424 (1981).
9. H. Samet, Bibliography on quadtrees and related hierarchical data structures, in *Data Structures for Raster Graphics*, F. J. Peters, L. R. A. Kessener, and M. L. P. van Lierop (Eds.), Springer-Verlag, Berlin, 181–201 (1986).
10. H. Samet, An overview of quadtrees, octrees, and related hierarchical data structures, in *Theoretical Foundations of Computer Graphics and CAD*. NATO ASI Series, Vol. F40, R. A. Earnshaw (Ed.), Springer-Verlag, Berlin, 52–68 (1988).
11. G. M. Hunter and K. Steiglitz, Operations on images using quadtrees. *IEEE Trans. on Pattern Analysis and Machine Intelligence* **1**(2), 145–153 (April 1979).
12. C. L. Jackins and S. L. Tanimoto, Octrees and their use in representing three-dimensional objects. *Comp. Graphics and Image Processing* **14**, 249–270 (November 1980).
13. C. L. Jackins and S. L. Tanimoto, Quadtrees, octrees, and k-trees: A generalized approach to recursive decomposition of Euclidean space. *IEEE Trans. on Pattern Analysis and Machine Intelligence* **5**(5), 533–539 (September 1983).
14. L. J. Doctor and J. G. Torborg, Display techniques for octree-encoded objects. *IEEE Comp. Graphics and Appl.* **1**(3), 29–38 (July 1981).
15. D. J. Meagher, Geometric modeling using octree encoding. *Comp. Graphics and Image Processing* **19**(2), 129–147 (June 1982).
16. M. Yau and S. N. Srihari, A hierarchical data structure for multidimensional images. *Comm. of ACM* **26**(7), 504–515 (1983).
17. M. Tamminen and H. Samet, Efficient octree conversion by connectivity labeling. *ACM Comp. Graphics* (*Proceedings of SIGGRAPH'84*) **18**(3), 43–51 (July 1984).
18. K. Yamaguchi, T. L. Kunii, K. Fujimura, and H. Toriya, Octree-related data structures and algorithms. *IEEE Comp. Graphics and Appl.* **4**(1), 53–59 (January 1984).
19. H. Samet, Region representation: Quadtrees from binary arrays. *Comp. Graphics and Image Processing*, 88–93 (May 1980).
20. H. Samet, Region representation: Quadtrees from boundary codes. *Comm. of ACM* **23**(3), 163–170 (March 1980).
21. H. Samet, An algorithm for converting rasters to quadtrees. *IEEE Trans. on Pattern Analysis and Machine Intelligence* **3**(1), 93–95 (January 1981).
22. C. A. Shaffer and H. Samet, Optimal quadtree construction algorithms. *Comp. Vision, Graphics, and Image Processing*, 402–419 (March 1987).
23. M. Potmesil, Generating octree models of 3D objects from their silhouettes in a sequence of images. *Comp. Vision, Graphics, and Image Processing*, 1–29 (October 1987).
24. S. D. Roth, Ray casting as a method for solid modeling, Technical Report GMR-3466, Computer Science Dept., General Motors Research Labs, Warren, Michigan (1980).
25. I. Gargantini, Translation, rotation, and superposition of linear quadtrees. *International Journal of Man-Machine Studies* **18**(3), 253–263 (March 1983).

64-7