Pattern Recognition Letters 10 (1989) 315–320
North-Holland

November 1989

Ray representation for k-trees

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

Wm. Randolph FRANKLIN
Department of Electrical, Computer, and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA

Received 25 March 1989

Abstract: k-trees have established themselves as useful data structures in pattern recognition. A fundamental operation regarding k-trees is the construction of a k-tree. We present a method to store an object as a set of *rays* and an algorithm to convert such a set into a k-tree. The algorithm is conceptually simple, works for any k, and builds a k-tree from the rays very fast. It produces a minimal k-tree and does not lead to intermediate storage swell.

Key words: Hierarchical data structures, representation of rigid solids, rays, stacking.

1. Introduction

k-trees are data structures based on the symmetric recursive partitioning of space. 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 that our results will be applicable to both quadtrees ($k = 2$) and k-trees with $k > 3$ as well.

We are concerned in this paper with a task of fundamental importance in systems based on octrees. This is the operation of constructing an octree. The algorithms cited in recent overviews [1,2] are neither efficient nor conceptually easy to understand and implement.

We aim to accomplish two things: (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. In a previous publication [3], we have called the rays *parallelepipeds*. It will be seen that the latter term is indeed more suggestive but for the sake of conciseness we shall adopt the former.

2. Terminology

An *octree* is a special case of the abstract data type *digital search tree* [5]. 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. Υ is made of 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 either be homogeneous (i.e., either fully occupied by Σ or void) or heterogeneous (i.e., partly occupied by Σ). We further subdivide the heterogeneous octants into suboctants. This is stopped when octants (possibly voxels) of uniform properties are obtained.

Following the established usage, let us consider the *obels*. An obel can be *empty*, *full*, or *partial*. 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 assume, without loss of generality, that the partial voxels, if any, at level-ℓ are arbitrarily declared to be full.

The level of an 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.

3. Relevant research

Quadtrees, octrees, and to a lesser extent higher dimensional 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 [6] and Srihari [7] for two informative surveys on the representation of rigid solids.

Early work on quadtree algorithms was done by Hunter and Steiglitz [8] who showed how to perform a set of operations on images using quadtrees. Jackins and Tanimoto [9] extended this work to octrees. Doctor and Torborg [10] presented display techniques for octree-encoded objects.

Construction of a software system to represent and manipulate irregular 3D objects was first accomplished by Meagher [11]. His system lets the user build quadtrees interactively, extend them to octrees, and carry out transformations and set-theoretic operations on them. Later, Meagher built a special hardware, 'The Solids Engine', which is a system for interactive solid modeling based on octrees.

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 [12]; 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 superficial resemblance between the algorithm in [12] and our algorithm. Tamminen and Samet [13] give an algorithm for converting from the boundary representation of a solid to the corresponding octree model using a technique called 'connectivity labeling'. Samet [14] considers the conversion of rasters to quadtrees.

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 from various viewpoints. These silhouettes are then processed to create a bounding volume that serves as an approximation of the object, cf. Potmesil [15].

4. Ray representation

Ray representation has, in fact, been known for a long time. For example, to compute the volume of a 3D object, Σ, one approximates it with a set of rectangular parallelepipeds. 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. Figure 1 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.

Let each ray be given in the 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 to a ray with fixed (x, y) that enters the object at z_1 and leaves it at z_2 (assume that $z_1 \leqslant z_2$). 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 the sequel, we sometimes ignore the surface normals since they can be easily handled by our algorithm.

Figure 1. Ray representation for a region (region boundary not shown).

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.)
- 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

Before we present our algorithm to build an octree from a set of rays, we consider the problem of obtaining a ray representation for quadric solids, e.g. an ellipsoid. Thus, we shall effectively show how to obtain an octree from a quadric solid, since the output rays from the following process are direcly supplied as input to the stacking algorithm of Section 7. It is fair to note that quadric solids are taken as an exemplary case for they are neither too trivial nor too complicated (like bicubic patches).

The algorithm below was implemented in Fortran-77 on a Prime 750.

Let Ξ be a 4×4 matrix and let $\mathscr{V}$ be the vector $[x\ y\ z\ 1]^T$. (The superscript T denotes the transpose.) A concise way of writing down the equation of a quadric is then $\mathscr{V}^T \Xi \mathscr{V} = 0$. We first clip a quadric solid to the unit cube (i.e., $[0,1]^3$) and then output a set of rays and surface normals as described before. We assume that each ray goes through the lower left corner of the obel; the alternative would have been that the rays go through the center.

The algorithm to obtain rays from quadric solids is as follows:

Algorithm

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. This range contains up to two (0, 1, or 2) segments that may be finite or infinite. To determine the range of x for which the 2D conic $f(x,z)$ is real, six cases should be considered depending on the discriminant of the formal solution for z in terms of x. That is, if $f(x,z) = az^2 + (b_0 + b_1x)z + (c_0 + c_1x + c_2x^2) = 0$, then the discriminant is $\delta(x) = (b_0 + b_1x)^2 - 4a(c_0 + c_1x + c_2x^2)$. The six cases are: $\delta(x)$ is always positive; $\delta(x)$ is positive outside a finite interval; $\delta(x)$ is positive inside a finite interval; $\delta(x)$ is never positive; $\delta(x)$ is positive above a certain value; and finally, $\delta(x)$ is positive below a certain value.

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 usual way. If z_i were clipped in step 2, then the associated normals would be $(0, 0 \pm 1)$.

The above algorithm is 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.

For ray tracing higher order solids, such as bicubic patches, we propose to use the following observation. The functions $f(x,y)$ and $n(x,y)$ — for intersections and normals — although complicated, are smooth. Therefore, efficient simple approximations using splines can be found.

6. Combining and splitting

Now we come to the main concern of the paper. 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 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, Figure 2 shows the nine maximal rows obtained from the ray $(1, 1, 17, 93)$.

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

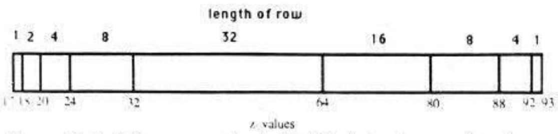

Figure 2. Splitting a ray (1, 1, 17, 93) into nine maximal rows (not to the scale).

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 (Figure 3).

Let $\rho_1, \rho_2, \ldots$ 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 (Figure 4).

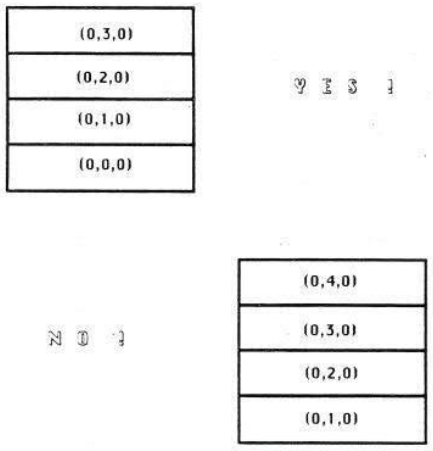

Figure 3. Four rows of length four that are combined into one square versus four rows that cannot be combined, since they are misaligned.

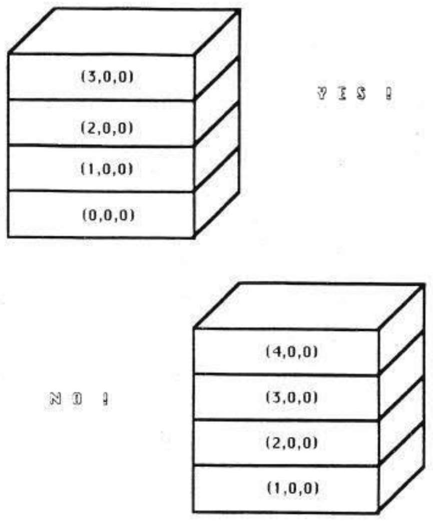

Figure 4. Four squares of length four that are combined into one cube versus four squares that cannot be combined, since they are misaligned.

Let $\sigma_1, \sigma_2, \ldots$ 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 generizable to cubes and hypercubes, once we are at $\geqslant$ 4D. See Figure 5 for illustrations of splitting.

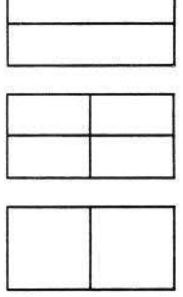
Figure 5. Two rows of length four that cannot be combined into a square but can be split into four rows of length two and combined into two squares of side two.

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 and these maximal rows have already been inserted into the relevant 1D $\Delta\Lambda$-lists.

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.

This process will build the octree, Ω, in its reduced form. (An octree is in its *reduced* form if it has no partial nodes with all empty or all full children.)

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. 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.

Algorithm

0. Each ray is partitioned 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.

1. The rows are sorted into lexicographic order by (y,z,x). This is necessary to detect combinable rows in one-pass.

2. Adjacent rows are combined into squares whenever this is possible. A square has a side of 2^l for some $0 \leqslant l \leqslant \ell$ and starting x and z coordinates that are multiples of 2^l. If we find 2^l rows with positions

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

where $m = 0, \ldots, 2^l$, then we can combine them into one square. Combining rows is illustrated in Figure 3. The process of combining and 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. Figure 5). 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. Next the squares are either combined into obels or split into squares of size 1 that are voxels. Combining squares is illustrated in Figure 4.

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].

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

We have implemented the stacking algorithm in Ratfor, a structured dialect of Fortran, 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 higher 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 [12], and Tamminen and Samet [13].

9. Conclusion

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 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. Thus, 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.

Acknowledgement

The second author's research is supported by the National Science Foundation under PYI grant number DMC-8351942.

References

[1] H. Samet and R. E. Webber, Hierarchical data structures and algorithms for computer graphics — Part I: Fundamentals. *IEEE Computer Graphics 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 Computer Graphics Appl.* 8 (4), 59–75 (July 1988).

[3] Wm. R. Franklin and V. Akman, Building an octree from a set of parallelepipeds. *IEEE Computer Graphics Appl.* 5 (10), 58–64 (October 1985).

[4] Wm. R. Franklin and V. Akman, Octree data structures and creation by stacking, in: N. Magenat-Thalmann and D. Thalmann, eds., *Computer-Generated Images: The State of the Art*. Springer, Tokyo, 176–185 (1985).

[5] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, MA (1973).

[6] A. A. G. Requicha, Representation of rigid solids: theory, methods, and systems. *ACM Comput. Surveys* 12 (4), 437-464 (December 1980).

[7] S. N. Srihari, Representation of three-dimensional digital images. *ACM Comput. Surveys* 13 (4), 400–424 (1981).

[8] G. M. Hunter and K. Steiglitz, Operations on images using quadtrees. *IEEE Trans. Pattern Anal. Machine Intell.* 1 (2), 145–153 (April 1979).

[9] C. L. Jackins and S. L. Tanimoto, Quadtrees, octrees, and k-trees: A generalized approach to recursive decomposition of Euclidean space. *IEEE Trans. Pattern Anal. Machine Intell.* 5 (5), 533–539 (September 1983).

[10] L. J. Doctor and J. G. Torborg, Display techniques for octree-encoded objects. *IEEE Computer Graphics Appl.* 1 (3), 29–38 (July 1981)

[11] D. J. Meagher, Geometric modeling using octree encoding. *Computer Graphics and Image Processing*. 19 (2), 129–147 (June 1982).

[12] M. Yau and S. N. Srihari, A hierarchical data structure for multidimensional images. *Comm. ACM* 26 (7), 504–515 (1983).

[13] M. Tamminen and H. Samet, Efficient octree conversion by connectivity labeling. *ACM Computer Graphics (Proc. SIGGRAPH '84)*, 18 (3), 43–51 (July 1984).

[14] H. Samet, An algorithm for converting rasters to quadtrees. *IEEE Trans. Pattern Anal. Machine Intell.* 3 (1), 93–95 (January 1981).

[15] M. Potmesil, Generating octree models of 3D objects from their silhouettes in a sequence of images. *Computer Vision, Graphics, and Image Processing*, 1–29 (October 1987).