Octrees for Faster Isosurface Generation*

Jane Wilhelms and Allen Van Gelder
University of California, Santa Cruz

Abstract

The large size of many volume data sets often prevents visualization algorithms from providing interactive rendering. The use of hierarchical data structures can ameliorate this problem by storing summary information to prevent useless exploration of regions of little or no *current* interest within the volume. This paper discusses research into the use of the *octree* hierarchical data structure when the regions of current interest can vary during the application, and are not known *a priori*. Octrees are well suited to the six-sided cell structure of many volumes.

A new space-efficient design is introduced for octree representations of volumes whose resolutions are not conveniently a power of two; octrees following this design are called *branch-on-need octrees* (BONOs). Also, a caching method is described that essentially passes information between octree neighbors whose visitation times may be quite different, then discards it when its useful life is over.

Using the application of octrees to isosurface generation as a focus, space and time comparisons for octree-based versus more traditional "marching" methods are presented.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory — semantics; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic — logic programming, model theory; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving — logic programming, nonmonotonic reasoning and belief revision

General Terms: Languages, Theory

Additional Key Words and Phrases: negation as failure, well-founded models, fixpoints, unfounded sets, stable models, three-valued logic

Authors' Addresses: Jane Wilhelms, Computer and Information Sciences Dept., Room 225AS, University of California, Santa Cruz, CA 95064, USA
Allen Van Gelder, Computer and Information Sciences Dept., Room 225AS, University of California, Santa Cruz, CA 95064, USA

*This research was supported in part by a State of California Micro-Electronics Grant, a UCSC Committee on Research Grant, NSF grant CCR-8958590, and a NASA-Ames Research Center Cooperative Agreement, Interchange No. NCA2-430.

1 Introduction

Interactive visualization is of major importance to scientific users, but the sheer size of volume data sets can tax the resources of computer workstations. Intelligent use of data structures and traversal methods can make a significant difference in algorithm performance. In particular, the use of hierarchical data structures to summarize volume information can prevent useless traversal of regions of little interest. However, the storage and traversal of hierarchical data structures themselves can add to the resource consumption of the algorithm, both in terms of time and space.

We are exploring the advantages and disadvantages of hierarchical data structures for visualization. In particular, we have explored the use of *octrees* in conjunction with a cell-oriented isosurface generation algorithm [27, 13, 26].

Octrees are particularly appropriate for representing sample data volumes common to scientific visualization, where the data points often define a spatial decomposition into hexahedral, space-filling, non-overlapping regions. Use of octrees for controlling volume traversal is appropriate whether regions are regular hexahedra (cubes, rectangular parallelopipids), as is common in medical imaging, or the irregular, warped hexahedra (curvilinear decompositions) that are common in computational fluid dynamics.

A volume whose maximum resolution is between 2^{k-1} and 2^k can be represented by an octree of depth k. This paper discusses the use of summary information at each node for the entire subvolume beneath it, making it possible to explore the volume contents without examining every data point. For isosurface generation the summary information consists of the maximum and minimum values of data within each node's region.

1.1 Background and Prior Work

Octrees, like quadtrees, are hierarchical data structures based on decomposition of space [15, 22, 16, 17, 14, 21, 20]. Quadtrees are two-dimensional decompositions that had their beginnings in the hierarchical representation of digital image data and spatial decomposition for hidden surface elimination [24, 11, 19]. In quadtrees, space is recursively subdivided into four subregions, hence the name "quad". Octrees are three-dimensional extensions of quadtrees, where space is recursively subdivided into eight subvolumes, and the root of the octree refers to the entire volume [15, 22, 16, 17]. In the normal case, each coordinate direction is divided in two, giving a "lower" half space and an "upper" half space. The effect of all three divisions is to create octants.

Octrees have been used to represent three-dimensional objects [10, 28]. Octrees also have been used just to represent the spatial relationship of geometrical objects, making it relatively simple to accomplish

such operations as locating neighbors [18] and to traverse the volume from front to back for hidden surface removal [4, 25].

In many octree applications, including those mentioned so far, the octree is used to represent some boolean property of the points in the volume, or some property for which most of the points take on a null value that is specified *a priori*. In image-processing terminology, a point in the volume is "black" (in the object), or "white" (uninteresting). Here we briefly review some storage optimizations that have been developed for such cases, and discuss why they do not carry over to the applications we have, in which the volume data can assume many values (none of which may be "uninteresting" *a priori*).

When the property is boolean, only one bit per octree node is needed. Levoy described a straightforward implementation for abstracting the (boolean) property of *nontransparency* from medical image data as part of volume rendering [12]. Initially, his method rounds the volume resolution up to 2^dx2^dx2^d, and assigns 8 data points to each node in the lowest level of the octree. It represents every node at the same level of the octree in a long bit-vector (1 = "black"), where 1 denotes that *some* child has value 1, or at the lowest level, that some data point is nontransparent among the 8 covered by the octree node. All octree information is located by address calculations; no pointers are needed. The storage overhead is acceptable, well under 20% of the original volume data in practice.

An alternative strategy is to prune lower portions of the octree when their values can be inferred from an ancestor [29]. One method is to define an internal node as "white" or "black" if all of its descendants are of that color, in which case no storage is allocated to the descendants; this process is called *condensation*. Otherwise the node is gray and has 8 explicit children. (For static nonboolean properties, only white nodes can be condensed.) How many octree nodes are needed depends on the original data. Because of the irregular shapes possible in such octrees, the structure must be represented explicitly, with pointers being the usual choice. Eight pointers per node use up storage quickly, so this implementation is workable only when the object can be represented with relatively few black and white nodes. However, it is possible to reduce the storage requirement to one pointer per node if all eight children of a node are allocated contiguously.

Linear octrees were introduced by Gargantini as a way to improve on the storage requirements of condensed, pointer-based octrees [6]. Related linear structures were used by others [16, 23]. Essentially, each "black" node in the condensed octree is assigned a key that encodes the path in the octree from the root to that node (see Section 4.1). Gray and white nodes are not allocated any storage, and the keys of the black nodes are stored in sorted order in one array (hence the name "linear"). Whether a linear octree requires more or less storage space than a bit-vector octree depends on the coherence of the boolean property being represented.

Glassner describes an implementation related to *linear octrees*, but with several innovations [8]. He uses a hash table instead of a sorted array to speed up node location by key. His ray tracing application requires storage of gray nodes, so he uses a slightly different key and allocates all 8 children of a node contiguously, so they can all be accessed under one key entry.

Bloomenthal also uses an octree to organize nonboolean data for implicit surface modeling [3]. A closed-form function is defined over the volume and evaluated by adaptive sampling. A piecewise polygonal representation is derived from the octree. The octree only pertains to the current isovalue, or threshold value, so this application also falls into the category of those whose data has a frequently occurring null value.

We are concerned here with the use of octrees to organize nonboolean data, where the points of interest cannot be determined *a priori*; that is, there is no frequently occurring null value. The reason that the condensation methods just discussed are not applicable in this context soon becomes evident: condensation occurs only when all children of a node have the same value, an event that may *never* occur in volumetric data such as density fields. We are not dealing with an object, or small set of objects, which occupies a possibly small portion of the volume, but rather a function that is defined throughout the volume. As we shall show in Section 3, without the benefits of large scale condensation, obvious octree designs can easily lead to prohibitive storage overhead.

Globus has independently investigated the use of an octree for isosurface generation [9]. His work is compared with ours in more detail in Sections 3.4 and 6.2. Briefly, he solved the storage problems by stopping the octree construction at a higher level, allowing an octree node to cover as many as 32 data points.

An alternative tree data structure for 3-dimensional data is the *3-d tree*, which is a special case of the k-d tree for k-dimensional data. A k-d tree is really a binary tree in which each node divides in some coordinate direction [2, 21]. The choice of direction can depend on the region "covered" by the node. By choosing to split the root in the z direction, to split the nodes at depth one in the y direction, those at depth two in the x direction, and repeating that cycle, we can simulate an octree with a 3-d tree. 3-d trees might offer advantages similar to octrees in some situations.

1.2 Summary of Results

Unlike earlier octree applications, where certain regions of the volume could be classified as uninteresting *a priori*, we studied the use of octrees to organize volume information when all of the volume is potentially interesting. In Section 4 we present an octree design that has proven to be efficient in time with acceptable

storage requirements, which we call a *branch-on-need octree* (BONO). Appendices A and B provide a technical supplement to this section. In Section 3 we show that more obvious alternate designs will have prohibitive storage requirements in most practical cases, where the resolutions of the volume are not precisely 2^dx2^dx2^d, as is usually conveniently assumed.

An important time-saver in isosurface generation is the re-use of computed information on cell edges that intersect the isosurface; each such edge is incident on four cells, so the computation can be used four times if it can be saved and located. A normal coordinatewise (marching) traversal of the volume permits a straightforward caching strategy with arrays [13]. However, the octree traversal order complicates storage-efficient caching considerably. We solved the problem with a hash table, as described in Section 5 and Appendix C. A key feature of the solution is that we can tell when a hash table entry has been retrieved for the last time, and delete it, making room for later entries.

Section 6 presents our experimental results, which compare the performance of an octree-based isosurface generation program with the more standard, non-hierarchical methods, such as marching cubes [13], and its variants [26]. In applying octrees to isosurface generation, it is important to remember that the only part of the processing that we are addressing is the detection and bypassing of trivial cells: those that do not intersect the current isosurface. Isosurface patches are calculated in significant cells with the same subroutines as used by the marching traversal, so their time cost is unchanged. Therefore, it is somewhat surprising that our experiments demonstrated speedups by factors of 2 and 3 in some cases, even when octree creation time is included.

Because one application often generates many isosurfaces from the same data, the speedup on the surface extraction phase alone is often more significant to the user. We observed speedups in the range of 1.6 to 11.

We develop a performance model based on the experimental data to predict the time requirements of isosurface generation with our implementations of both octree traversal and marching traversal.

Rounding out the paper, Section 2 reviews polygon-based isosurface generation methods, and describes our adaptations of previous methods to take advantage of an octree, and Section 7 draws some conclusions and suggests future directions for the research.

2 Octrees in Conjunction with Isosurface Generation

A common approach to visualization is to extract a geometrical representation of a surface of constant threshold value (the *isosurface*) from sampled volume data. Graphics workstations are deft at handling such geometrical representations efficiently, offering the ability to render hundreds of thousands of polygons per second. For large volumes with complex surfaces, however, generation of the geometric representation may

take many minutes.

A popular method for isosurface generation is to imagine the volume as consisting of cells whose corners are the sample values [27, 13, 3, 5, 26]. Each cell is examined one by one for the presence of an isosurface, which is detected when at least one corner value is above and another below the threshold value. If the isosurface intersects the cell, intersection points along the cell edges are calculated and become the vertices of polygons representing the portion of the isosurface within that cell. Lorensen and Cline introduced a table-lookup method to speed polygon generation [13].

Generally, isosurfaces intersect a small subset of the cells within a volume. However, most of the useful work of the algorithm occurs within those cells that do intersect the isosurface. The relative costs of traversal versus cellular computation are extremely variable, depending upon the total size of the volume, the number of cells including the isosurface, and the size of the computer memory. Previous research indicated that between 30% and 70% of the time spent in isosurface generation was spent examining empty cells [26]. This provided impetus for the study of the octree traversal methods described here.

2.1 Two Contrasting Approaches

We explored two approaches to isosurface generation: the first is a typical *marching* method [13, 26] and the second is the *octree-traversal* method introduced here. Both methods read the volume data into an array, begin with a setup phase, then continue with a surface-finding phase for each threshold furnished by the user.

The marching method has a minimal setup phase; for the user's convenience in selecting thresholds, it finds the maximum and minimum data values. Each surface-finding phase visits all cells of the volume, normally by varying coordinate values in a triple "for" loop. As each cell that intersects the isosurface is encountered, the necessary polygons to represent the portion of the isosurface within the cell are generated. There is no attempt to "trace" the surface into neighboring cells. To find the isosurface for a new threshold value the whole phase is repeated; there is no carry-over information.

During its setup phase, the octree method creates an octree that contains at each node the maximum and minimum data values found in that node's subtree. The lowest level of the octree represents eight cells, and contains a pointer into the data array to the sample value having the minimum (x, y, z) value of any of the 27 samples defining these eight cells. The volume data is stored in an ordinary 3-D array, rather than octree traversal order, to simplify the location of neighboring data points. In contrast to the marching method, the setup phase does a substantial amount of work, and determining maxima and minima are an essential part of the setup, not merely a user convenience.

In surface-finding phases, the octree is traversed with a particular threshold, only exploring those branches that contain part of the isosurface; any node whose maximum is below the threshold or whose minimum is above it is exited without traversing its children. When a leaf node that contains isosurface is visited, each of the (normally 8) cells that it "covers" are visited, and polygons are generated.

Both methods use a table lookup for polygon generation. The basic idea is due to Lorensen and Cline [13]; refinements to handle "ambiguous" cells were described by Wilhelms and Van Gelder [26]. The table contains 256 entries, referring to the 256 combinations of positive and negative (relative to threshold) values that can occur for an eight-cornered cell. Each table entry describes which cell edges contain intersections and how they should be joined to produce the polygons representing the isosurface. A second table is used to treat ambiguous cases; each ambiguous case in the first table contains a "pointer" to the relevant section of the second table.

3 Space Requirements of Previous Octree Designs

The space requirements of an octree can be a serious issue in the design of a system that will process large data volumes. In this section we examine space requirements of previous designs; in Section 4 we describe a more space-efficient design. First, we review octree basics and introduce some terminology.

3.1 Octree Basics

Octrees are tree structures of degree eight. It is convenient to number the children from 0 to 7; their numbers, written in binary, encode which sub-region of the parent they "cover". We shall use the zyx convention. If the z bit is 1, the child covers an octant that is "upper in z"; if it is 0, the child covers an octant that is "lower in z". The y and x bits are similarly interpreted. We shall write child numbers in binary to facilitate this interpretation. (An alternate notation is back/front for z, south/north for y, and west/east for x.)

Traversal of an octree is accomplished by recursively visiting a node and traversing its children in order. Notice that all children of a fixed node that are "lower in z" are visited before all children that are "upper in z"; among those that are in the same z division, the ones that are "lower in y" are visited first, etc.

A *full octree* is one in which each node has exactly eight children; however, this is possible only if the volume's resolution is the same power of two in each dimension, e.g. 4x4x4 (64) samples, 8x8x8 (512) samples, 64x64x64 (262,144) samples, etc. Full octrees offer the best ratio of the number of nodes to data points. For a volume with a resolution of s in each direction where s is a power of 2,

$$nodes = \sum_{i=0}^{\log_2 s - 1} 8^i = \frac{s^3 - 1}{7}$$

As described in Section 3.3, the regularity of the full octree data structure permits it to be implemented without storing explicit addressing information in the octree node; we call this a *pointerless* octree design. An alternative is to use an octree design in which nodes are allocated space only if they "cover" a region that is actually within the volume. This makes the location of nodes less predictable. Traditionally, each node contains addressing information necessary to determine the location of its children. We call this design a *pointer* octree.

3.2 A Running Example

Previous treatments of octrees have made the simplifying assumption that the resolutions of the volume are precisely 2^dx2^dx2^d for some integer d. However, power-of-two volumes are not the norm; moreover, volumes often vary widely in resolution among the three dimensions. In this case, storage of a full octree can be extremely wasteful because many nodes correspond to regions not actually within the volume.

To explore the impact on previous designs when the power-of-two assumption does not hold, we shall consider an example at some length. To make the discussion concrete, let us assume that each data value requires the same space as a pointer or index, and call this a "word". Usually a "word" is 32 bits. For our application, isosurface generation, each octree node must store two words (*maximum* and *minimum*), plus whatever structural book-keeping is required.

For our running example, consider a data volume whose x, y, and z resolutions are 320x320x40, for a total of 4,096,000 data points. An octree for this volume will have nine levels. This is the size of one of our CT-scan volumes for which computational experience is presented; see Section 6.

3.3 Pointerless Full Octree Design

Suppose we naively set up a "full" octree over this 320x320x40 volume; that is, every node in the octree has exactly 8 children, and each leaf node refers to 8 data points (which may or may not be within the actual volume). One motivation for using a full octree is that the nodes can be stored in an array T in such a way that parent and child pointers are not required. In analogy with the heap-sort strategy in one dimension and the pyramid strategy in two dimensions, the root is in $T[0]$, the children of the node occupying $T[k]$ are found in locations $T[8k+1] \ldots T[8k+8]$, and all nodes at the same level are contiguous within the array. (Thus we can think of $T[9] \ldots T[72]$ as a subarray containing all the nodes at depth two, etc.)

Unfortunately a full octree for the example volume requires

$$1 + 2^3 + 4^3 + \ldots + 256^3 = 19,173,961$$

nodes of two words each, or almost 40 million words. (As discussed earlier, in applications requiring only one

Node depth	Number of nodes	Region covered
0	1	320x320x40
1	8	160x160x20
2	64	80x80x10
3	512	40x40x5
4	4,096	20x20x3 or 20x20x2
5	24,576	10x10x2 or 10x10x1
6	98,304	5x5x2 or 5x5x1
7	393,216	3x3x2, 3x2x2, 2x2x2 or 3x3x1, 3x2x1, or 2x2x1
8	786,432	2x2x2, 2x2x1, or 2x1x1
total	1,307,209	

Editor please note: figures also are given at end of text.

Figure 1: A even-subdivision octree covering a 320x320x40 data volume.

bit per node the full octree fits in slightly over 500,000 words, which is quite acceptable.) This overhead in this example, almost 4 times as many nodes as data points and nearly 8 times as much space as the original volume, is almost certainly not acceptable. Most of the space is wasted, but is not easy to eliminate because "real" nodes are scattered throughout it.

3.4 Traditional Design of Pointer Octrees

An alternative is to build the octree in the more traditional way, with pointers or indices (subscripts) to a node's children within the node record. Nodes are only created if they "cover" some portion of the data. Leaf nodes should require only one pointer, which is to data, because the neighbors of that point can be located. Although a naive design would specify a pointer for each child, giving at least 8 pointers per internal node, with some care, all of a node's children can be allocated contiguously, so that one child pointer (or index) suffices for internal nodes as well. In this design, each octree node occupies 3 words. (Some designs might include a fourth word for a parent pointer.)

Let us see how this might work, following the traditional and intuitive *even-subdivision* strategy, which divides each node's range from the top down in each coordinate direction as evenly as possible. (Ranges of 1

or 2 are not divided.) Consider an octree in which the three resolutions are not equal. Because nodes branch in each dimension from the top down until no more subdivisions are required, the even-subdivision strategy will result in 8-way branching at the top when all dimensions subdivide, 4-way branching in the middle when two dimensions subdivide, and (effectively) binary subtrees at the bottom when only the largest dimension continues to subdivide. As most nodes are at the greater depths, this means the least efficient nodes are the most numerous; consequently, the ratio of nodes to data points can be quite high.

The outcome for our example is shown in Figure 1. Observe that a node has null children whenever one or more of its coverage resolutions is 1 or 2. (Strategies to detect when this occurs are not difficult, and are similar to those employed in our actual implementation, as discussed later.) When all coverage resolutions are 2 or less, the node is a leaf, and it points to data.

As Figure 1 shows, an octree designed by this strategy for our example consists of about 1,300,000 nodes, and nearly 4 million words. This octree requires almost as much memory as the original volume – better than a full pointerless octree, but still a serious overhead. Even if a pointerless strategy were devised for this octree, the use of even subdivision will produce an octree whose size is about 2/3 of the original volume. The ratio of octree nodes to data points is 0.3191, a significant degradation when compared to the optimum of 0.1428. This observation motivated our search for an improvement.

Globus reports a variation of the even-subdivision strategy that addresses the storage space issue [9]. Primarily, a coarser granularity is accepted in that an octree node that covers less than 32 cells is not further subdivided. Also, nodes that cover small but very oblong regions are divided 4 or 8 times in the longest dimension, and not divided in one or both of the shorter dimensions. On our example Globus' strategy yields an octree of 201,289 nodes; each leaf node turns out to cover 25 cells in a 5x5x1 pattern. The number of words needed per node depends on implementation choices that were not reported; various trade-offs between time and space are possible. Comparison with his timing results appears in Section 6.2.

4 A Space-Efficient Octree Design

This section describes the octree design we adopt and compares the space requirements with the even-subdivision method. Essentially, we regard the octree as conceptually full, but avoid allocating space for empty subtrees. Note that the even-subdivision strategy divides the volume, from the top down, whenever it can. The approach we describe now, in some sense, *delays* subdivision until absolutely necessary. Therefore we call it the *branch-on-need* strategy, and we call the resulting data structure a *branch-on-need octree* (BONO for short). The presentation here is from a top-down point of view, because the procedures work top-down to facilitate storage allocation. An alternative bottom-up view is discussed in Section 4.2.

Node	Conceptual Region			Actual Region		
	x	y	z	x	y	z
root	0–511	0–511	0–511	0–319	0–319	0–39
000 child	0–255	0–255	0–255	0–255	0–255	0–39
001 child	256–511	0–255	0–255	256–319	0–255	0–39
010 child	0–255	256–511	0–255	0–255	256–319	0–39
011 child	256–511	256–511	0–255	256–319	256–319	0–39
100 child	0–255	0–255	256–511	0–255	0–255	*empty*
...	...	...	...	...	...	*empty*

Editor please note: figures also are given at end of text.

Figure 2: Conceptual and Actual Regions for octree nodes over a 320x320x40 volume.

4.1 Branch-on-Need Octrees

We can associate with each node a conceptual region and an actual region, as illustrated in Figure 2 with our running example, which is a 320x320x40 volume. Recall that in our terminology, "001 child" means the "lower z, lower y, upper x" child. The three bits of the child code represent motion in the z, y, and x directions, respectively, when read left to right. Since all of the "upper z" children of the root have empty actual regions, no space is allocated for them. Therefore the root has only 4 actual children, and we say that it "branches" in the x and y directions, but not in the z direction.

A further element of the BONO strategy is that the "lower" subdivision in each branching direction always covers the largest possible exact power of two (yielding a range of the form $2^k - 1$). A two-dimensional analog contrasting the even-subdivision and branch-on-need strategies is shown in Figure 4 on a 5x6 array.

Definition 4.1: We define the *range* in each of the x, y, and z directions as the difference between the upper and lower index limits of the actual region. The *range vector* is the triple of x, y, and z ranges. □

An interesting patterns emerges if we look at a node's range vector in binary. In our example, for the root we have:

Direction	Range in Binary	(Decimal)
x	100111111	(319)
y	100111111	(319)
z	000100111	(39)

The directions that branch are precisely those that have a 1 bit in the leftmost position, when all ranges are written with the same number of bits. The number of bits equals the height of the node in the octree, with leaves considered height 1 (and data considered height 0). To obtain the range of the "upper" children in a direction that branches, simply remove that leftmost 1 bit. The range of the corresponding "lower" children is a bit string of 1's, one shorter than the root's original range. Following this rule, the ranges of the 001 child of the root are:

Direction	Range in Binary	(Decimal)
x	00111111	(63)
y	11111111	(255)
z	00100111	(39)

We see immediately that this node branches in the y direction, but not the x or z directions. Both of its children will have the following range configuration:

Direction	Range in Binary	(Decimal)
x	0111111	(63)
y	1111111	(127)
z	0100111	(39)

The bit patterns of the ranges also allow us to quickly discover how many (actual) nodes the octree has at each depth. This information is vital for the allocation of storage. For example, to see how many nodes are at depth 4, we take the 4 leftmost bits of each range of the root:

Direction	Range in Binary
x	1001|11111
y	1001|11111
z	0001|00111

This gives ranges of 9, 9, and 1. Add 1 to each and multiply, giving $10 * 10 * 2 = 200$ nodes at depth 4. To see why this works, just imagine that we *began* with a data volume of resolutions 10x10x2, and built an octree over it. Then the root's ranges would be 9, 9, and 1.

Figure 3 shows the number of nodes produced by this strategy on our 320x320x40 example (4,096,000 data points). In contrast to previous schemes, the 585,439 nodes are far fewer than the number of data points, and the ratio is virtually the optimum one of 1 node per 7 data points for a full octree. The space, about 1,750,000 words, is well under 50% of that occupied by the data volume. This behavior is typical, and not an artifact of the resolutions chosen for the example. As shown rigorously in Appendix A, when all

Node depth	Number of nodes	Region covered
0	1	320x320x40
1	4	256x256x40 or 256x64x40 or 64x256x40 or 64x64x40
2	9	128x128x40 or 128x64x40 or 64x128x40 or 64x64x40
3	25	64x64x40
4	200	32x32x32 or 32x32x8
5	1,200	16x16x16 or 16x16x8
6	8,000	8x8x8
7	64,000	4x4x4
8	512,000	2x2x2
total	585,439	

Editor please note: figures also are given at end of text.

Figure 3: Nodes by depth for a branch-on-need octree (BONO) covering a 320x320x40 volume.

resolutions are at least 32, the ratio of octree nodes to data points never exceeds 0.1615, which is not far from the optimum of 0.1428.

Using the above observations, we can efficiently allocate precisely the correct amount of space for an octree, and determine in which directions any given node branches. Our implementation precomputes this information and stores in each octree node a 3-bit code to tell which directions branch, and an index to its "leftmost" (000) child. The actual children of the node are contiguous in the array in lexicographic order. That is, if all 8 children are actual, their order is 000, 001, 010, 011, 100, 101, 110, 111. We assume the volume has less than 2^{28} data points and pack the code and index into one 32-bit word.

In fact, all nodes at a given depth are contiguous and appear in what we call *shuffled zyx* order. For example, at depth 4, a node's origin in the data volume is the triple

Direction	Origin in Binary
x	$x_9x_8x_7x_6000000$
y	$y_9y_8y_7y_6000000$
z	$z_9z_8z_7z_6000000$

where x_9 is 0 for the lower-in-x children of the root, and is 1 for the upper-in-x children of the root, etc.

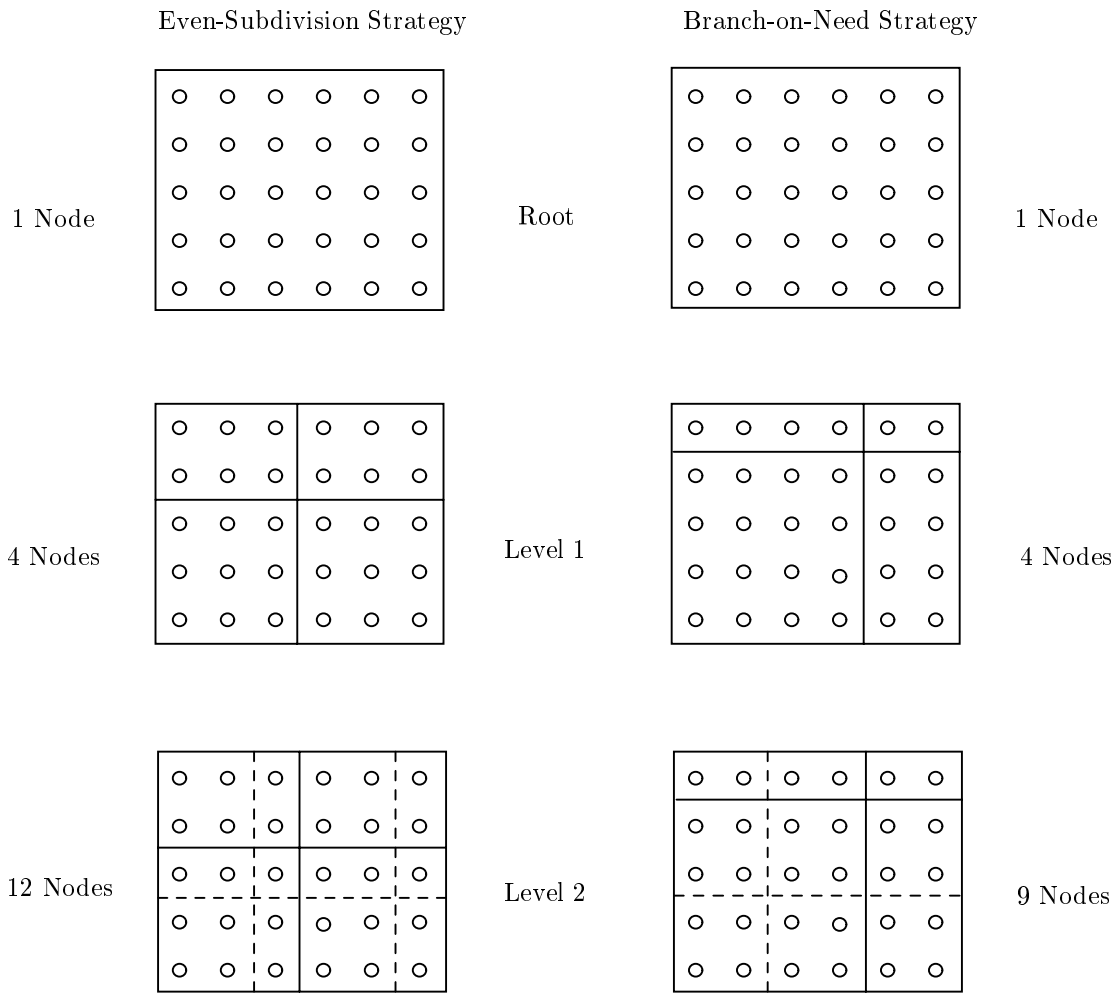
The node's *shuffled zyx* key is

$$z_9y_9x_9z_8y_8x_8z_7y_7x_7z_6y_6x_6$$

Interpreting the bits of this key in groups of three gives the path in the octree to this node.

Appendix B describes how to calculate the location of a node without using pointers (or indices) from its key and the range vector of the root of the octree. Although this can be done in time proportional to the length of the key, it is still fairly expensive, so we chose to incur the space overhead of one index per node to speed traversal of the octree. As the discussion showed, this space overhead is about 15% of the basic data volume. Furthermore, it can be greatly reduced by use of pointerless nodes on the leaf level only.

Our isosurface application does not require the ability to locate an arbitrary node in the octree, but this is a necessary operation for many other algorithms [6, 8, 23, 7]. For example, Gargantini uses (essentially) the same shuffled zyx key to specify a node's place in the octree [6]. However, her *linear octree* explicitly stores the key with the node, and stores the nodes sorted in key order; a binary search is employed to locate a node in memory by its key. Glassner uses a slightly different key and a hash table [8]. In contrast, we do not store the key at all, but can calculate the node's location in memory from the key.



Editor please note: figures also are given at end of text.

Figure 4: Comparison of designs in two dimensions on a 5x6 array.

4.2 Comparison of Branch-on-Need and Even-Subdivision Strategies

The branch-on-need strategy can also be viewed as a *bottom-up* one, as compared to the top-down even-subdivision strategy described earlier. Data points are grouped from the bottom up in the most efficient manner. For volumes with different resolutions in the three dimensions, this places the 8-way branching section of the hierarchy (or 8-way collapsing if one thinks from the bottom up) at the bottom of tree; 4-way branching occurs where one dimension has been clustered as much as possible; and an (effectively) binary tree occurs where only one dimension still requires clustering. Thus, the most efficient reductions take place at the bottom of the tree where there are the most nodes. In general, this strategy is space-wise far superior

to the even-subdivision approach. Figure 4 demonstrates a two-dimensional analog of this phenomenon on a 5x6 array; in three dimensions and with larger resolutions the difference is much more pronounced. The two methods are identical and produce a full octree when the volume resolutions are precisely 2^dx2^dx2^d.

Two factors influence the ratio of nodes to data points for branch-on-need octrees: the size of each resolution (resolutions of a power of two are best, and of a power of two plus one are worst); and the relative size of the resolutions (cubical volumes are best). The size of each resolution is a more significant influence, but neither effect seriously degrades the ratio. We show this in an intuitive manner below; Appendix A gives the formal proof. Consider volumes all of whose resolutions are at least 32. (This bound is chosen as a reasonable minimum resolution. The true worst case ratio can be produced by having two resolutions of 1. The tree is then a binary tree, which is uninteresting.)

First, consider a volume that contains $(s+1)$x$(s+1)$x$(s+1)$ data points, where s is a power of two. A full octree can be built over sxsxs data points, which will contain $\frac{1}{7}(s^3-1)$ nodes. Three quadtrees can be built over the remaining nodes covering three faces; each one will contain $\frac{1}{3}(s^2-1)$ nodes. Three binary trees can be built along the three edges which still include nodes not considered, each adding approximately s nodes. The necessity of nodes with only two or four children on the deepest levels of the tree produces the worst case, which is approximately $\frac{1}{7}s^3+s^2$ nodes actually needed, compared with the optimum of about $\frac{1}{7}(s+1)^3$. While not a major degradation, the ratio of actual to optimum is about $(1+4/s)$.

For larger volumes, up to a resolution of $2s$x$2s$x$2s$, the "binary" and "quad" nodes in the above described tree can be given new children, making them more "efficient", and the ratio of nodes to data points will never be worse than for the above case. For each power of two over 32, the ratio improves slightly, approaching $\frac{1}{7}$. Thus, the worst case among cubical volumes is one with resolutions 33x33x33. This volume requires 5803 nodes for 35937 data points, a ratio of .1615.

What about the effect of one resolution being much larger than the other? Consider the effect of a volume of resolutions sxsxt, where s is a power of two and at least 32, as before, and $t >> s$. A octree of height $\log_2 s$ can be build over the volume. At the top of this octree, two resolutions cannot undergo further divisions and the other resolution is s. A binary tree of height $\log t - \log_2 s$ can be built on top of the octree. The octree contains approximately $\frac{1}{7}s^2t$ nodes and the binary tree approximately t/s nodes. The effect of the binary tree is relatively negligible; the ratio of nodes to data points remains near $\frac{1}{7}$.

The branch-on-need strategy normally produces a tree shaped quite differently from the even-subdivision version. The even-subdivision strategy will partition the volume into more equal parts. Using a one-dimensional example, if there are 65 data points, the even-subdivision strategy will divide at the top into one subtree covering 33 data points and another covering 32. The branch-on-need strategy will divide into

one region covering 64 data points and the other covering 1 point.

In rare cases, it may be necessary to visit more nodes in a BONO than in an even-subdivision octree to process a particular section of the volume. In this one-dimensional example, if the two highest-indexed points were the only ones of interest, the branch-on-need tree would require visitation of 13 nodes because these two points are on separate subtrees of the root. The even-subdivision tree would require only 6 or 7. However, there are counterexamples, as when the two points are in the center of the volume, where the even-subdivision tree requires more visitation because the nodes are on separate subtrees of the root while in the BONO tree they are not. Furthermore, as traversal has a modest cost (see Section 6), this whole issue is not a major consideration.

5 Recalling Previously Computed Intersection Points

Each vertex is generally contained in four neighboring polygons, and each vertex requires six floating point numbers: three representing location, and three a normal vector required for rendering. Calculating vertex information, particularly the normal vector, is quite expensive, and a substantial time savings is realized by re-using the results. Storage is also saved by representing polygons by indices into a vertex array, saving over twice the space normally required for geometric representations.

A straightforward array method for saving this information, as described by Lorensen and Cline [13], is used in the marching method. However, it is not suitable for octrees because the octree traversal does not visit the nodes in row-major order. It is possible to devise a savings method designed particularly for use with octrees, but a hash table appeared to provide a simple and general solution to the problem. Wyvill *et al.* also used a hash table in their implementation [27]. Technicalities of our hash table are given in Appendix C.

The main observation needed for storage efficiency here is that it is possible to identify the last visit of an edge, and remove its information from the hash table, freeing the storage for later use. Because of traversal order, the three edges adjacent to the "origin" of a cell will never be visited again. This allowed us to use hash tables which are much smaller than would be necessary to store all significant edges simultaneously. Artzy *et al.* used a related storage optimization in their traversal of an implicit binary spanning tree; although their data structure was a set of linked lists, the removal of "marked nodes" that would never be visited again was critical [1].

6 Experimental Results

Tests were run on six sets of data. In all cases, use of octree traversal was faster than the marching approach. This was true whether the time for octree creation was included or whether only the actual surface-finding traversal times were compared. The justifications for comparing the two methods on the surface-finding phases only are two: first, the octree can be precomputed and stored for reuse; and, second, it is possible to use the same octree for multiple thresholds.

6.1 Description of Experimental Data

Table 1 describes the data. The *blunt fin* (C. M. Hung and P. G .Buning, NASA Ames Research Center) is a curvilinear volume generated using computational fluid dynamics and extracting a surface from the density field. The superoxide dismutase *enzyme* (D. McRee, Scripps Clinic) and the high-potential iron *protein* (L. Noodleman and D. Case, Scripps Clinic) are molecular volumes from the volume data set distributed by the University of North Carolina. The *dolphin* (T. Cranford, UC Santa Cruz) is a threshold from a CT-scan of a dolphin head, using only central slices. The *MR-brain* (Siemens Medical Systems) and the *CT-head* (North Carolina Memorial Hospital) are also scans from the UNC dataset. Figure 5 shows the images of some of these surfaces generated on a Silicon Graphics Iris.

6.2 Experimental Timing Results

Table 2 summarizes the costs of the two methods on the seven sets of data. Runs were made on a Sun Sparcstation 1 with 8 megabytes of memory. Tests on a 16 megabyte machine have produced similar relative timing results. Note that, in general, octree traversal shows increasingly better relative performance as the data files get large. This is to be expected, as the fraction of cells containing isosurface tends to decrease as volume size increases.

The *octree-creation* time includes allocating memory for the octree, recursively traversing it downwards establishing pointers, and accumulating maximum and minimum information on the return to the root node. The *surface-finding* time involves, for the marching version, traversal of all cells, and, for the octree version, traversal of the regions of the octree and volume as dictated by summary information in the nodes. The *total generation* time is the sum of these for the octree method, and is the same as the surface-finding time for the marching method. The time to display the resultant polygons would be the same for both methods, and is not included. The time to read in the data and do minor preliminary initialization is not included in the statistics either, because it is the same for both methods. It is worth mentioning that for very large files, this cost is approximately equal to the octree creation time.

Data File	Resolution	Number of Cells	Octree Size	Threshold	% Cells with Surface
Blunt Fin	40x32x32	31,117	5,855	1.0	12.97%
Protein	64x64x64	226,981	37,449	0.1	0.82%
Enzyme	97x97x116	998,468	160,383	36.5	9.22%
Dolphin	320x320x40	3,718,093	585,439	120.2	2.87%
NMR Brain	256x256x109	6,784,954	1,032,229	500.5	7.04%
CT Head	256x256x113	7,040,990	1,070,373	150.5	4.28%

Editor please note: tables also are given at end of text.

Table 1: Characteristics of Data Volumes

In the best experimental case, the protein, the octree method improved surface-finding speeds by a factor of 11. For this volume, relatively few cells have surfaces and these are concentrated in small regions of the volume. For the other volumes, surface-finding using the octree took between a quarter and two thirds the time of the standard method.

The actual octree traversal times were insignificant. For example, the MR Brain volume took under 4.5 seconds for actual octree traversal, compared to almost 300 seconds for the surface-finding phase.

Globus reports experiments on the same blunt fin data set that we used [9]. He does not report precise thresholds used, but the threshold we used should be most comparable to his maximum case. There we both found that the octree produces a substantial time reduction in the surface-finding phase: he reports about 4.7 *vs.* 8.9, or 53% to compare with our 64%. However, he experiences a somewhat larger relative cost for building the octree: 1.78/8.9, or 20% of the "marching" time. We found it to be .36/3.00, or 12%. This difference may be due to a more complicated subdivision strategy (see Section 3.4). In any event we both observe a benefit even when only one surface is extracted, with increasing dividends as the octree is reused for subsequent surfaces. Globus, like us, found substantially greater speedups in other cases, including some by a factor of 9.9.

6.3 Performance Models of Surface Finding

We estimated a linear function of the total number of cells and the number of cells intersecting the isosurface to explain the running times observed in our experiments. Specifically, the running time is modelled as $time = At + Bs$, where t is the total number of cells actually visited and s is the number with isosurface. The

	March Traversal	Octree Traversal	% Octree/March
Blunt Fin			
Octree Creation		0.36	
Surface Finding	3.00	1.90	64%
Total Extraction	3.00	2.26	75%
Protein			
Octree Creation		2.50	
Surface Finding	11.1	0.95	9%
Total Extraction	11.1	3.45	31%
Enzyme			
Octree Creation		11.9	
Surface Finding	75.2	43.0	57%
Total Extraction	75.2	54.9	73%
Dolphin			
Octree Creation		56.5	
Surface Finding	224.9	60.7	27%
Total Extraction	224.9	117.2	52%
MR Brain			
Octree Creation		109.6	
Surface Finding	556.5	282.2	51%
Total Extraction	556.5	391.8	70%
CT Head			
Octree Creation		114.1	
Surface Finding	464.1	167.1	36%
Total Extraction	464.1	281.2	61%

Editor please note: tables also are given at end of text.

Table 2: Comparative processing times for isosurface generation, in CPU seconds.

Editor please note: figures also are given at end of text.

Figure 5: Selection of Isosurfaces analyzed.

estimates of constants A and B are shown below. The value of A for the octree includes the overhead of traversing the internal nodes.

Surface-finding times in μ-secs.	march	octree
per cell visited (A)	54	72
additional time if cell intersects isosurface (B)	431	431

We do not have enough data to achieve statistical significance; these values were informally estimated, and fit the larger runs better than the small ones.

We observed that the octree method very consistently visited about twice as many cells as had isosurface, because the lowest internal node indicates whether an isosurface may be present in any of up to eight cells. Thus, $t = 2s$ for the octree method, to a good approximation. Of course, t equals the whole volume for marching methods. From this crude model we can estimate a *ratio* of surface-finding times of the two methods as a function of f, the *fraction* of cells that intersect the isosurface. Let r be the ratio of marching time to octree time for the surface-finding phase. Let t now be the total number of cells in the volume; thus $s = ft$. We have

$$r = \frac{54t + 431ft}{72(2ft) + 431ft} = .75 + \frac{.094}{f}$$

Whenever f is less than about .37, r exceeds 1, and we anticipate that the octree method will outperform the marching methods *in the surface-finding phase*. We have yet to encounter a situation where the percentage of cells with surface is anywhere near this.

The other side of the coin is that the octree method requires significant setup time, other than reading in the data. For this time factor, we estimated $C = 16$ μ-seconds per cell in volume. (The comparative value for the marching method is just the time per cell to get the maximum and minimum of the volume, and is optional; it is about 5 μ-seconds.)

As remarked before, the setup time can often be amortized over several surface-finding phases. However, for a run consisting of setup and finding one isosurface, we get the ratio r_1 marching time to octree time for the run:

$$r_1 = \frac{5t + 54t + 431ft}{16t + 72(2ft) + 431ft} = .75 + \frac{.082}{f + .028}$$

Whenever f is less than about .30, r_1 exceeds 1, and we anticipate that the octree method will outperform the marching methods for single isosurface runs (including both setup and surface-finding phases). For f less than about .037, we calculate that octrees are better by a factor of 2.

6.4 Miscellaneous Remarks

Use of an octree traversal without a saving strategy to re-use previous intersection-point-related computations obviated its speed advantages. Preliminary results showed the octree traversal was about equivalent to the traditional marching method when the octree did not save those computations. Similarly, removal of the saving strategy from the traditional method significantly slows down that algorithm. Therefore, use a saving strategy.

Another interesting side note is that traversal times on machines with relatively small memories can be highly dependent on traversal order. Assume the x dimension varies fastest in the array that stores the data volume, followed by y, and then z. We inadvertently found ourselves at one time traversing the volume in the order z-y-x and found that it could take many times as long as an x-y-z traversal, due to the time taken by page faults. Our particular tests were on an 8 megabyte machine, which is surely "relatively small memory" by current standards in graphics. But small memory is relative: Moving to a machine with greater capacity soon leads us to work with larger volumes. While there is no reason to use z-y-x order for polygon generation methods, algorithms such as direct volume rendering normally access the volume front to back, and hence direction can make a significant difference. In such cases, storing the data in octree order to equalize traversal costs might be preferred.

Even very small data sets stored in ascii take a long time to read and convert. Ascii is most convenient for portability, but should be converted to floats or integers, as appropriate, for repeated use.

7 Conclusions and Future Research

Our studies showed that octrees can yield substantial improvements in performance for isosurface generation on data sets produced by current technologies. These improvements will be even more significant on larger data sets. However, several technical problems had to be solved to realize the benefits:

1. A new octree implementation made the storage overhead acceptable.

2. A careful caching method enabled us to re-use results of earlier computations, then discard them when they would not be referenced again, freeing the storage for others.

Numerous topics remain to be explored concerning octrees in scientific visualization. The use of octrees on irregular, non-hexahedral grids, as are often produced in finite-element analysis, requires further research. Other applications of octrees should also present new, interesting, technical problems.

Acknowledgements

We wish to thank Peter Hughes for his seminal suggestion concerning the use of min-max hierarchies and Judy Challinger for her initial implementation of the marching approach, which served as the starting point for this work. Our work also benefited from discussions with Marc Levoy, Nelson Max, and Brian Wyvill. We wish to thank Ted Cranford for his CT-scan data of dolphins, NAS/NASA-Ames Research Center for computational fluid dynamics data, and University of North Carolina for their very useful volume data set. We wish to thank the reviewers for their many helpful comments. We also thank Silicon Graphics Incorporated, Digital Equipment Corporation, and Sun Microsystems for their generous gifts of equipment, without which this research would not have been possible.

Appendix

A Ratio of Branch-on-Need Octree Nodes to Data Points

In this appendix we show that the BONO design creates a number of nodes that is less than $\frac{1}{6}$ the number of data points covered, for volumes all of whose dimensions are at least 32. The lower bound of 32 was chosen as a bound that should be satisfied in practice. The same analysis can be repeated with looser restrictions to get slightly weaker bounds. Recall that $\frac{1}{7}$ is the best possible case, and is easily achieved by all methods when the volume is 2^dx2^dx2^d.

To simplify the formulas, we use the nomenclature introduced in Definition 4.1 that the *range* in each coordinate direction is one less than the number of points (or resolution) in that direction. Thus a 32x32x48 volume has the range vector $(31, 31, 47)$.

Let (x_0, y_0, z_0) denote the range vector for the data. Let (x_h, y_h, z_h) denote the range vector for the octree nodes at height h, where the leaves of the octree are at height 1. As discussed earlier, from the way

the octree is formed,

$$x_h = \lfloor x_0/2^h \rfloor$$
$$y_h = \lfloor y_0/2^h \rfloor$$
$$z_h = \lfloor z_0/2^h \rfloor$$

That is, h rightmost bits are dropped off each range coordinate at height zero.

Throughout this discussion let d be the height of the root, and let $1 \leq h \leq d$. We are concerned with bounding the ratio

$$R(x_0, y_0, z_0) = \sum_{h=1}^{d} \frac{(x_h+1)(y_h+1)(z_h+1)}{(x_0+1)(y_0+1)(z_0+1)}$$

for ranges $x_0 \geq 31$, $y_0 \geq 31$, $z_0 \geq 31$. Thus the minimum *volume* to which our bound applies is 32x32x32.

To find the point where $R(x_0, y_0, z_0)$ achieves its maximum, we use the fact that R must be maximized in any coordinate direction when the other two coordinates are held constant. The analysis is complicated by the fact that there are numerous local maxima. To simplify the formulas that follow, we abbreviate:

$$K_h = \frac{(y_h+1)(z_h+1)}{(x_0+1)(y_0+1)(z_0+1)}$$

First we consider points such that x_0 is a power of 2, and compare $R(2^m, y_0, z_0)$ for different values of m.

Lemma A.1: $R(2^m, y_0, z_0)$ is a decreasing function of m, for $1 \leq m < d$ and fixed y_0 and z_0.

Proof: $R(2^{m+1}, y_0, z_0) - R(2^m, y_0, z_0)$ can be written as

$$\sum_{h=1}^{m} \left(\frac{(2^{m+1-h}+1)}{(2^{m+1}+1)} - \frac{(2^{m-h}+1)}{(2^m+1)} \right) K_h$$
$$+ \left(\frac{2}{(2^{m+1}+1)} - \frac{1}{(2^m+1)} \right) K_{m+1}$$
$$+ \sum_{h=m+2}^{d} \left(\frac{1}{(2^{m+1}+1)} - \frac{1}{(2^m+1)} \right) K_h$$

The second sum is clearly negative (or zero if $m+1 = d$). The other terms can be simplified to

$$\sum_{h=1}^{m} \left(\frac{(2^{m-h} - 2^m)}{(2^{m+1}+1)(2^m+1)} \right) K_h$$
$$+ \left(\frac{1}{(2^{m+1}+1)(2^m+1)} \right) K_{m+1}$$

Using the facts that $m \geq 1$ and $K_{m+1} \leq K_m$, we see that the m-th term of the sum is at least as negative as the term under the sum is positive. Since all terms of the sum are negative, the lemma is proved. ∎

Next we show that a power of 2 dominates all the values up to the next power of 2.

Lemma A.2: $R(2^m, y_0, z_0) > R(2^m + k, y_0, z_0)$ for $1 \leq k < 2^m$, $1 \leq m < d$, and fixed y_0 and z_0.

Proof: $R(2^m + k, y_0, z_0) - R(2^m, y_0, z_0)$ can be written as

$$\sum_{h=1}^{m} \left(\frac{(2^{m-h} + \lfloor k/2^h \rfloor + 1)}{(2^m + k + 1)} - \frac{(2^{m-h} + 1)}{(2^m + 1)} \right) K_h$$
$$+ \sum_{h=m+1}^{d} \left(\frac{1}{(2^m + k + 1)} - \frac{1}{(2^m + 1)} \right) K_h$$

The second sum is clearly negative. The first sum can be simplified to

$$\sum_{h=1}^{m} \left(\frac{(-k2^{m-h} + 2^m \lfloor k2^{-h} \rfloor - k + \lfloor k2^{-h} \rfloor)}{(2^m + k + 1)(2^m + 1)} \right) K_h$$

Since all terms of the sum are negative or zero, the lemma is proved. ∎

Now we can establish a worst-case bound on the ratio of BONO nodes to data points:

Theorem A.3: For $x_0 \geq 31$, $y_0 \geq 31$, and $z_0 \geq 31$, the maximum value of $R(x_0, y_0, z_0)$ occurs at $(32, 32, 32)$, corresponding to a 33x33x33 data volume.

Proof: Lemmas A.1 and A.2 show that no $x_0 > 32$ can produce the maximum, and direct calculation shows $R(31, y_0, z_0) < R(32, y_0, z_0)$, so R must be maximized at $x_0 = 32$. The same arguments apply to y_0 and z_0. ∎

The ratio of BONO nodes to data points for a 33x33x33 volume is

$$R(32, 32, 32) = \frac{17^3 + 9^3 + 5^3 + 3^3 + 2^3 + 1}{33^3}$$
$$= .1615 < \tfrac{1}{6}$$

which is the worst case for volumes all of whose dimensions are at least 32.

B Location of Nodes from Keys

Here we sketch the calculation of a node's location, given its depth in the octree, its shuffled zyx key, and the range vector of the octree. The underlying idea is that nodes at the same depth preceding the desired node are in the subtree of a smaller sibling of some ancestor of the desired node. The relationship to Gargantini's *linear octrees* is discussed near the end of Section 4.1.

Assume the key is a list of octal digits describing the path from the root to the desired node. Let function `head` return the first element of such a list, and let `tail` return the list of all elements except the first. Also,

```
offset(key, range, depth)
    {
    count = 0;
    if (depth > 0)
        {
        ancestor = head(key);
        for (sibling = 0; sibling < ancestor; sibling++)
            {
            s = childRange(sibling, range);
            count += (s.x + 1) * (s.y + 1) * (s.z + 1);
            }
        count += offset(tail(key), childRange(ancestor, range), depth-1);
        }
    return count;
    }
```

Figure 6: Procedure to calculate node location from shuffled zyx key.

let `childRange(child,range)` return the range vector of a child, given the child number (an octal digit), and `range`, the range vector of the parent, as described in Section 4.

The function `offset(key, range, depth)` shown in Figure 6 returns the number of actual nodes at depth `depth` whose shuffled zyx keys are lexicographically less than `key` in an octree whose range vector is `range`. This number gives the offset of the desired node from the beginning of the subarray of nodes at that depth.

The depth of recursion equals the depth of the desired node, which is proportional to the length of the key. The work at each level is bounded by a constant, as the `for` loop body executes at most 7 times. Thus the function takes time that is linear in key length. Nevertheless, a substantial time penalty would be incurred if this calculation were used instead of a pointer, to save space.

C Hash Table Design Details

This appendix describes some details of the hash table used to store edge-related calculations for later use in the octree traversal, as discussed in Section 5. Other designs are certainly workable, but this can provide a starting point for other implementors.

Each edge in the volume is given a unique key. Assume the edge is from (x, y, z) to a point one greater in some coordinate direction. Then the offset of (x, y, z) in the volume array (viewed now as one-dimensional) is the basis for the key. The "direction-code" assigned is 1 for x, 2 for y, or 3 for z. The key is then

$$4 * \text{offset} + \text{direction-code}$$

The "null" key is 0.

As an example, consider the edge from $(3, 3, 3)$ to $(3, 4, 3)$ in a 320x320x40 volume. The offset of $(3, 3, 3)$ is 308,163, and this is a y edge, so its key is 1,232,654.

Our experience showed that a good size for the hash table is 8 times the square root of the number of data points in the volume, rounded up to a power of 2. We wanted a fast hash function that would distribute edges in the same cell. We settled on one shown diagrammatically in Figure 7; its mathematical form is

$$(K/F) \oplus 8(K \bmod F)$$

where $\oplus$ denotes *exclusive or*, K is the key and F, the "fold point", is 1/8 the size of the hash table. (This function relies on K being less than $8F^2$.) Upon collision, we rehashed by adding 1 mod table size.

We tracked utilization of the hash table, and found that it normally got about one quarter full, and averaged about one collision per operation.

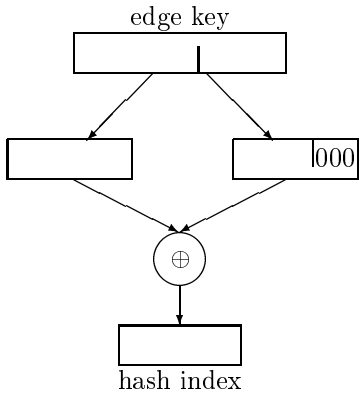


Figure 7: The hash function, where "⊕" denotes "exclusive or".

References

[1] E. Artzy, G. Frieder, and G. Herman. The theory, design, implementation, and evaluation of a three-dimensional surface detection algorithm. *Computer Graphics and Image Processing*, 15(1):1–24, January 1981.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):214–229, 1975.

[3] Jules Bloomenthal. Polygonization of implicit surfaces. *Computer-Aided Geometric Design*, 5:341–355, 1988.

[4] Louis J. Doctor and John G. Torborg. Display techniques for octree-encoded objects. *IEEE Computer Graphics and Applications*, 1(3):29–38, July 1981.

[5] Richard S. Gallagher and Joop C. Nagtegaal. An efficient 3-D visualization technique for finite element models. *Computer Graphics*, 23(3):185–194, 1989.

[6] I. Gargantini. Linear octrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, 20(4):365–374, December 1982.

[7] I. Gargantini, T. R. Walsh, and O. L. Wu. Viewing transformations of voxel-based objects via linear octrees. *IEEE Computer Graphics and Image Processing*, 6(10):12–20, October 1986.

[8] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[9] A. Globus. Octree optimization. In *Symposium on Electronic Imaging Science and Technology*, San Jose, CA., February 1991. SPIE/SPSE.

[10] C. L. Jackins and Stephen Tanimoto. Oct-trees and their use in representing 3D objects. *Computer Graphics and Image Processing*, 14:249–270, 1980.

[11] A. Klinger and C. R. Dyer. Experiments on picture representation using regular decomposition. *Computer Graphics and Image Processing*, 5:68–105, March 1976.

[12] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3), July 1990.

[13] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[14] X. Mao, T. Kunii, I. Fuhishiro, and T. Nomoa. Hierarchical representations of 2D/3D gray-scale images and their 2D/3D two-way conversion. *IEEE Computer Graphics and Applications*, 7(12):37–44, December 1987.

[15] D. J. Meagher. Octree encoding: A new technique for the representation, manipulation, and display of arbitrary three-dimensional objects by computer. Technical Report IPL-TR-80-111, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, N.Y., October 1980.

[16] D. J. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, 1982.

[17] D. J. Meagher. Interactive solids processing for medical analysis and planning. In *Proceedings NCGA 5th Annual Conference*, pages 96–106, Dallas, TX, May 1984.

[18] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. *Computer Graphics (ACM Siggraph Proceedings)*, 22(4):289–298, August 1988.

[19] Hanan Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):186–, June 1984.

[20] Hanan Samet. *Applications of Spatial Data Structures.* Addison-Wesley Publishing Company, Inc., 1990.

[21] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley Publishing Company, Inc., 1990.

[22] S. N. Srihari. Representation of three-dimensional digital images. *Computing Surveys*, 13(4):399–424, 1981.

[23] M. Tamminen and H. Samet. Efficient octree conversion by connectivity labeling. *Computer Graphics (ACM Siggraph Proceedings)*, 18(3):43–51, July 1984.

[24] J. E. Warnock. The hidden line problem and the use of halftone display. In M. Faiman and J. Nievergelt, editors, *Proceedings of the Second University of Illinois Conference on Computer Graphics, Pertinent Concepts in Computer Graphics*, pages 154–163, March 1969.

[25] Alan Watt. *Fundamentals of Three-Dimensional Computer Graphics.* Addison-Wesley Publishing Company, Reading, Mass., 1st edition, 1989.

[26] Jane Wilhelms and Allen Van Gelder. Topological ambiguities in isosurface generation. Technical Report UCSC-CRL-90-14, CIS Board, University of California, Santa Cruz, 1990. Extended abstract in ACM Computer Graphics 24(5) 79–86.

[27] G. Wyvill, C. McPheeters, and B. Wyvill. Data structures for soft objects. *The Visual Computer*, 2(4):227–234, August 1986.

[28] K. Yamaguchi, T. L. Kunii, and K. Fujimura. Octree-related data structures and algorithms. *IEEE Computer Graphics and Applications*, 4(1):53–59, January 1984.

[29] M. M. Yau and S. N. Srihari. A hierarchical data structure for multi-dimensional digital imaging. *Communications of the ACM*, 26(7):504–515, July 1983.

Node depth	Number of nodes	Region covered
0	1	320x320x40
1	8	160x160x20
2	64	80x80x10
3	512	40x40x5
4	4,096	20x20x3 or 20x20x2
5	24,576	10x10x2 or 10x10x1
6	98,304	5x5x2 or 5x5x1
7	393,216	3x3x2, 3x2x2, 2x2x2 or 3x3x1, 3x2x1, or 2x2x1
8	786,432	2x2x2, 2x2x1, or 2x1x1
total	1,307,209	

Figure 1

A even-subdivision octree covering a 320x320x40 data volume.

Node	Conceptual Region			Actual Region		
	x	y	z	x	y	z
root	0–511	0–511	0–511	0–319	0–319	0–39
000 child	0–255	0–255	0–255	0–255	0–255	0–39
001 child	256–511	0–255	0–255	256–319	0–255	0–39
010 child	0–255	256–511	0–255	0–255	256–319	0–39
011 child	256–511	256–511	0–255	256–319	256–319	0–39
100 child	0–255	0–255	256–511	0–255	0–255	*empty*
...	...	...	...	...	...	*empty*

Figure 2

Conceptual and Actual Regions for octree nodes over a 320x320x40 volume.

Node depth	Number of nodes	Region covered
0	1	320x320x40
1	4	256x256x40 or 256x64x40 or 64x256x40 or 64x64x40
2	9	128x128x40 or 128x64x40 or 64x128x40 or 64x64x40
3	25	64x64x40
4	200	32x32x32 or 32x32x8
5	1,200	16x16x16 or 16x16x8
6	8,000	8x8x8
7	64,000	4x4x4
8	512,000	2x2x2
total	585,439	

Figure 3

Nodes by depth for a branch-on-need octree (BONO) covering a 320x320x40 volume.

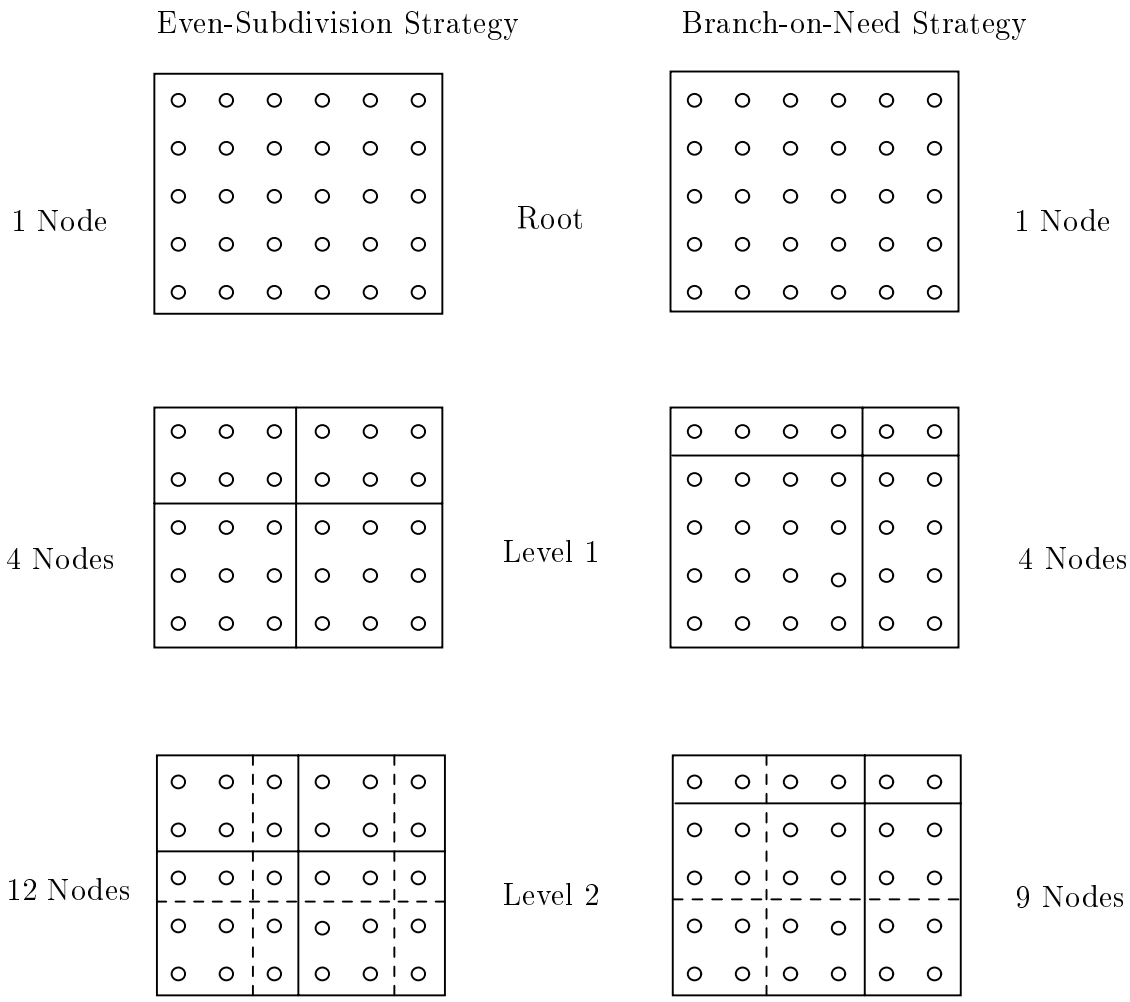
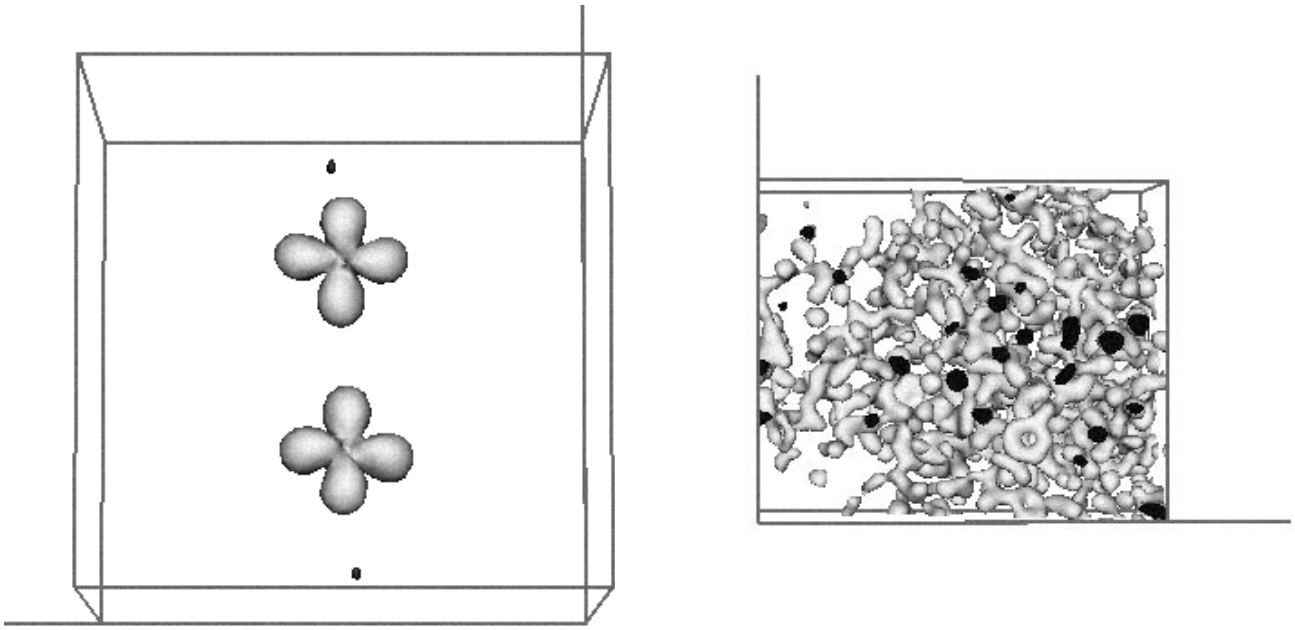
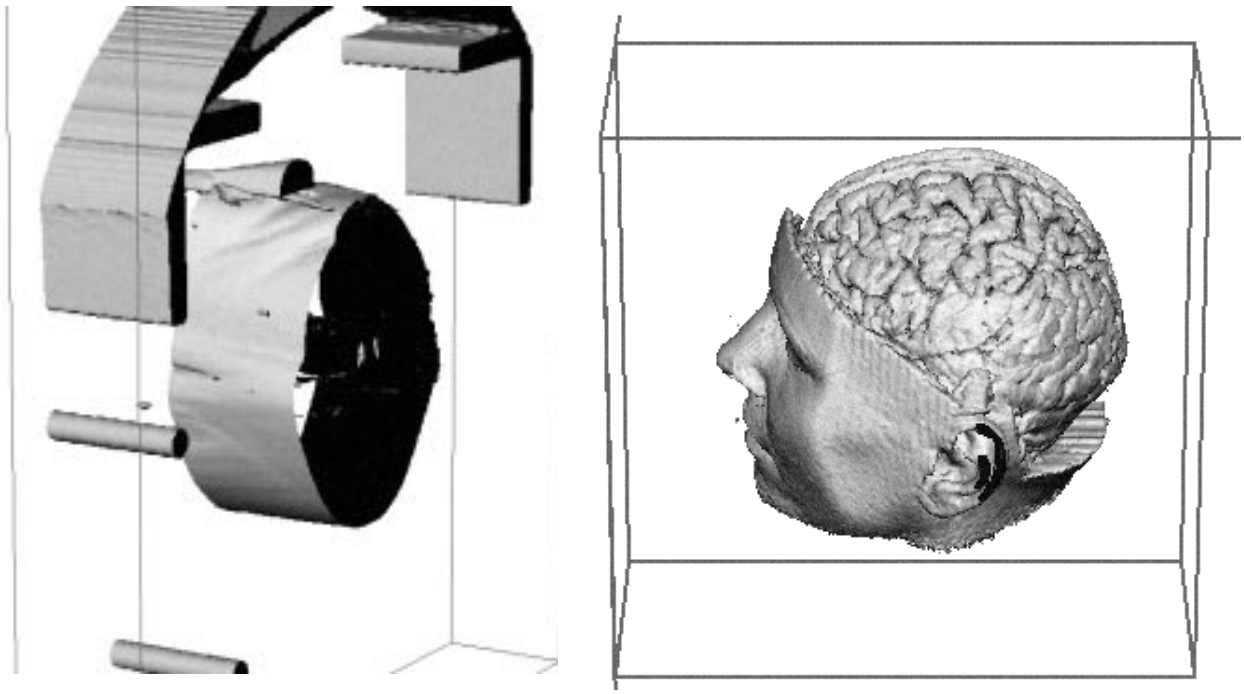


Figure 4

Comparison of designs in two dimensions on a 5x6 array.



a. Iron protein wave function threshold .1 **b.** Enzyme electron density map threshold 36.5



c. Dolphin threshold 120.2 **d.** NMR-scan of head threshold 500.5

Figure 5. Selection of Isosurfaces analyzed.

```
offset(key, range, depth)
    {
    count = 0;
    if (depth > 0)
        {
        ancestor = head(key);
        for (sibling = 0; sibling < ancestor; sibling++)
            {
            s = childRange(sibling, range);
            count += (s.x + 1) * (s.y + 1) * (s.z + 1);
            }
        count += offset(tail(key), childRange(ancestor, range), depth-1);
        }
    return count;
    }
```

Figure 6. Procedure to calculate node location from shuffled zyx key.

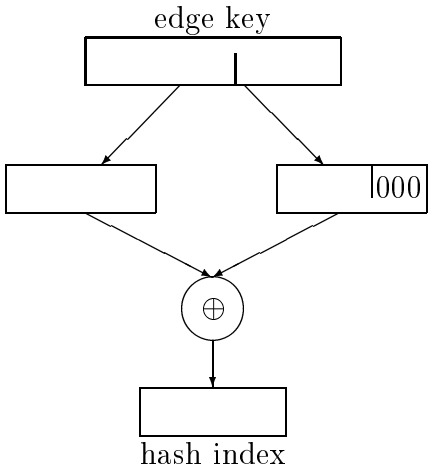


Figure 7. The hash function, where "$\oplus$" denotes "exclusive or".

Data File	Resolution	Number of Cells	Octree Size	Threshold	% Cells with Surface
Blunt Fin	40x32x32	31,117	5,855	1.0	12.97%
Protein	64x64x64	226,981	37,449	0.1	0.82%
Enzyme	97x97x116	998,468	160,383	36.5	9.22%
Dolphin	320x320x40	3,718,093	585,439	120.2	2.87%
NMR Brain	256x256x109	6,784,954	1,032,229	500.5	7.04%
CT Head	256x256x113	7,040,990	1,070,373	150.5	4.28%

Table 1

Characteristics of Data Volumes

	March Traversal	Octree Traversal	% Octree/March
Blunt Fin			
Octree Creation		0.36	
Surface Finding	3.00	1.90	64%
Total Extraction	3.00	2.26	75%
Protein			
Octree Creation		2.50	
Surface Finding	11.1	0.95	9%
Total Extraction	11.1	3.45	31%
Enzyme			
Octree Creation		11.9	
Surface Finding	75.2	43.0	57%
Total Extraction	75.2	54.9	73%
Dolphin			
Octree Creation		56.5	
Surface Finding	224.9	60.7	27%
Total Extraction	224.9	117.2	52%
MR Brain			
Octree Creation		109.6	
Surface Finding	556.5	282.2	51%
Total Extraction	556.5	391.8	70%
CT Head			
Octree Creation		114.1	
Surface Finding	464.1	167.1	36%
Total Extraction	464.1	281.2	61%

Table 2

Comparative processing times for isosurface generation, in CPU seconds.