# Accelerated Isosurface Extraction in Time-Varying Fields

Philip M. Sutton and Charles D. Hansen, *Member*, *IEEE*

**Abstract**—For large time-varying data sets, memory and disk limitations can lower the performace of visualization applications. Algorithms and data structures must be explicitly designed to handle these data sets in order to achieve more interactive rates. The Temporal Branch-on-Need Octree (T-BON) extends the three-dimensional branch-on-need octree for time-varying isosurface extraction. This data structure minimizes the impact of the I/O bottleneck by reading from disk only those portions of the search structure and data necessary to construct the current isosurface. By performing a minimum of I/O and exploiting the hierarchical memory found in modern CPUs, the T-BON algorithm achieves high performance isosurface extraction in time-varying fields. This paper extends earlier work on the T-BON data structure by including techniques for better memory utilization, out-of-core isosurface extraction, and support for nonrectilinear grids. Results from testing the T-BON algorithm on large data sets show that its performance is similar to that of the three-dimensional branch-on-need octree for static data sets while providing substantial advantages for time-varying fields.

**Index Terms**—Isosurface, time-dependent scalar field visualization, multiresolution methods, octree, bricking, unstructured grid visualization, out-of-core visualization.

## 1 Introduction

Researchers in many science and engineering fields rely on insight gained from instruments and simulations that produce discrete samplings of three-dimensional scalar fields. Visualization methods allow for more efficient data analysis and can guide researchers to new insights. Isosurface extraction is an important technique for visualizing three-dimensional scalar fields. By exposing contours of constant value, isosurfaces provide a mechanism for understanding the structure of the scalar field. These contours isolate surfaces of interest, focusing attention on important features in the data, such as material boundaries and shock waves, while suppressing extraneous information. Several disciplines, including medicine [1], [2], computational fluid dynamics (CFD) [3], [4], and molecular dynamics [5], [6], have used this method effectively.

Understanding the dynamic behavior of a data set requires the visualization of its changes with respect to time. However, most high performance computers possess neither the disk space nor the amount of memory necessary to store and manipulate large[1] time-varying data sets efficiently. While visualization research has begun to address this problem [7], [8], data sets from both computational and measurement sources have continued to increase in size, putting pressure on storage systems. Simulations that compute and store multiple time steps further increase the demand for storage space, commonly producing data sets on the order of one half to one gigabyte per time step with hundreds of time steps. With this vast amount of data to process, visualization programs may become slow and unwieldy, consuming large amounts of time reading multiple files from disk and performing swapping due to limited physical memory. Without a high degree of interactivity, the user loses the visual cues necessary to understand the structure of the field, reducing the effectiveness of the visualization.

We present an algorithm for isosurface extraction in time-varying fields that minimizes the impact of the I/O bottleneck. By reading only those portions of the data and search structure necessary to construct the current isosurface, the Temporal Branch-on-Need Octree (T-BON) makes efficient use of both I/O and memory, greatly accelerating isosurface extraction for large dynamic data sets. This work builds on a previous paper [9] and extends that work by presenting methods for improving memory behavior, isosurface extraction in curvilinear and unstructured grids, and out-of-core isosurface extraction.

In the following sections, we first discuss related work and then present our algorithm for extracting isosurfaces in time-varying fields. We introduce several improvements on the previously published T-BON algorithm which provide better performance and better generality. We then provide experimental results, demonstrating the performance of the algorithm on several large time-varying data sets. Finally, we draw conclusions and suggest directions for future work.

1. Our test data sets range in size from 8.4MB to 537MB per time step.

- *P.M. Sutton is with Lawrence Livermore National Laboratory, PO Box 808, L-561, Livermore, CA 94550. E-mail: psutton@llnl.gov.*
- *C.D. Hansen is with the Department of Computer Science, University of Utah, 50 S. Central Campus Dr., 3190 MEB, Salt Lake City, UT 84124. E-mail: hansen@cs.utah.edu.*

*Manuscript received 15 Mar. 2000; accepted 3 Apr. 2000.*
*For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 111481.*

## 2 Previous Work

A number of different techniques have been introduced to increase the efficiency of isosurface extraction over the

1077-2626/00/$10.00 © 2000 IEEE

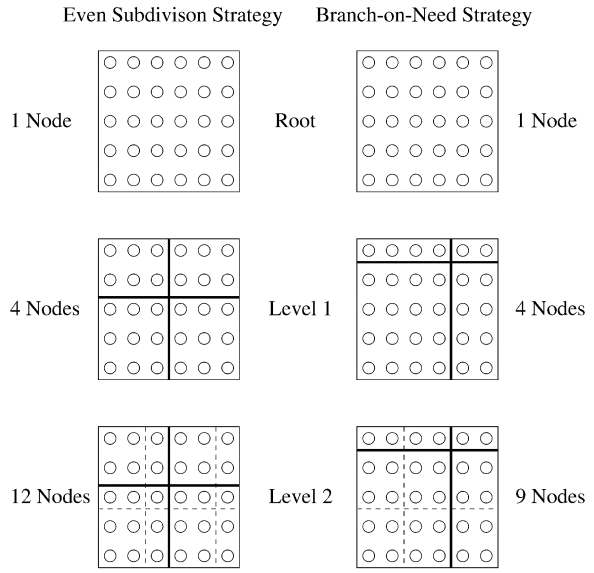



Fig. 1. Two-dimensional example of the branch-on-need algorithm (from Wilhelms and van Gelder [12]). The branch-on-need strategy produces fewer nodes when the dimensions of the data are not powers of two.

linear search proposed in the Marching Cubes algorithm [10], [11]. Wilhelms and van Gelder [12] describe the branch-on-need octree (BONO), a space-efficient variation on the traditional octree. Their data structure partitions the cells in the data based on their geometric positions. Extreme values (minimums and maximums) are propagated up the tree during construction such that only those nodes that span the isosurface, i.e., those with $minvalue < isovalue < maxvalue$, are traversed during the extraction phase.

The branch-on-need octree resembles the even-subdivision octree, but partitions the cells more efficiently when the dimensions of the volume are not powers of two. Fig. 1 compares the strategies in two dimensions. The even-subdivision strategy divides the volume in each direction at each level of the tree, while the branch-on-need strategy partitions the volume such that the "lower" subdivision in each direction covers the largest possible power of two cells. This results in fewer nodes, allowing the tree to be traversed more efficiently.

Recent methods have focused on partitioning the cells based on their extreme values. Livnat et al. [13] introduced the span space, where each cell is represented as a point in 2D space. The point's $x$-coordinate is defined by the cell's minimum value and the $y$-coordinate by the maximum value. The NOISE algorithm described in [13] uses a kd-tree to organize the points. Shen et al. [14] use a lattice subdivision of span space in their ISSUE algorithm. This simplifies and accelerates the search phase of the extraction, as only one element in the lattice requires a full min-max search of its cells. This acceleration comes at the cost of a less efficient memory footprint than the kd-tree.

The Interval Tree technique introduced by Cignoni et al. [15] guarantees worst-case optimal efficiency. Cells, represented by the intervals defined by their extreme values, are grouped at the nodes of a balanced binary tree. For any isovalue query, at most one branch from a node is traversed.

An alternate technique is to propagate the isosurface from a set of seed cells. Itoh et al. [16], [17], Bajaj et al. [18], and van Kreveld et al. [19] construct seed sets that contain at least one cell per connected component of each isosurface. The isosurface construction begins at a seed and is traced through neighboring cells using adjacency and intersection information.

An algorithm to improve I/O performance and allow efficient isosurface extraction on data sets larger than physical memory was described by Chiang et al. [7], [8]. An interval tree is built on disk using a two-level hierarchy. Cells are first grouped into meta-cells and a meta-interval defined. These meta-intervals are then composed into an interval tree, which is divided into disk block-sized groups to allow efficient transfer from disk.

Weigle and Banks [20] consider time-varying scalar data as a four-dimensional field. They construct an "isovolume" for each isovalue, representing the volume swept by the isosurface over time. Imposing a time constraint on the isovolume yields an instantaneous surface. This method elegantly captures temporal coherence, but its high execution time makes it impractical for large data sets.

Shen [21] proposed the Temporal Hierarchical Index Tree to perform isosurface extraction on time-varying data sets. This method classifies the data cells by their extreme values over time. Temporal variation of cells is defined using lattice subdivision, extending the ISSUE algorithm. Nodes in the tree contain cells with differing temporal variation and are paged in from disk as needed to extract an isosurface at a particular time step. At every time step, an ISSUE search [14] is performed at each node. In order to accelerate the full min-max search, an Interval Tree is constructed in those lattice elements that may require such a search. The Temporal Hierarchical Index Tree shows significant improvement in storage requirements over construction of a span-space search structure which treats each time step as an independent data set. It achieves this while retaining an efficient search strategy for isosurface extraction.

Shen's work clearly accelerates the search for isosurfaces in time dependent data. However, at each time step, the entire data domain (time step) is loaded into physical memory. The isosurface extraction process potentially needs to access all of the time steps in a time-varying data set. If all time steps do not simultaneously fit into physical memory, I/O can become a bottle neck. As noted by Wilhelms and Van Gelder [12], for a particular isovalue, large portions of the data not containing the isovalue need not be examined. Similarly, these same large portions of the data need not be read from disk when constructing an isosurface. For time dependent data sets, this savings can be significant and has led us to develop a method aimed at exploiting this observation.

## 3 Temporal Branch-on-Need Octree (T-BON)

To provide high interactivity for isosurface extraction, an algorithm's underlying data structure must use memory and I/O efficiently. Experimental comparisons [22] between

a number of three-dimensional isosurface algorithms show that the branch-on-need octree provides an efficient structure for isosurface extraction. The BONO has low memory overhead and exploits spatial coherence and memory layout to yield low execution times, making it ideal for performing isosurface extraction in time-varying fields.

Both time-varying algorithms discussed in Section 2 [20], [21] attempt to capture the temporal coherence of the isosurface. They reason that, given one isosurface at a given time step, a high probability exists that that isosurface will pass through the same neighborhood on the next time step. Data structures that utilize temporal coherence could predict where in the hierarchy to search on the next time step. However, in order to construct triangles, an isosurface extraction algorithm must interpolate along cell edges, using the data values at each vertex. Therefore, the data for the next time step must reside in memory before interpolation and isosurface construction can begin. If all time steps cannot fit in memory, the algorithm must fetch the data from disk, causing a bottleneck in the I/O system. The design of the T-BON data structure attempts to accelerate isosurface extraction by minimizing this bottleneck rather than by exploiting temporal coherence in the isosurface.

Sections 3.1 and 3.2 review the construction and traversal of the T-BON data structure, as presented in [9]. The original method reduces I/O latency, but does not fully exploit the I/O system design. Section 3.3 describes a method for further reducing the effect of the I/O bottleneck. Section 3.4 presents techniques for extending the T-BON data structure to include curvilinear and unstructured grids, generalizing the underlying octree structure, which was designed for rectilinear grids. Finally, Section 3.5 presents two out-of-core algorithms based on the T-BON data structure, allowing systems with limited memory to perform isosurface extraction on data sets larger than main memory.

### 3.1 Construction

A preprocessing step builds a branch-on-need octree for each time step in the data and stores it to disk in two sections. The information common to all trees is saved only once. This includes the general infrastructure of the tree, such as branching factors and pointers to children or siblings. This information can be created knowing only the size of the data in each dimension. Extreme values for the nodes are computed and stored separately, as these values can vary at each time step and the T-BON does not utilize temporal coherence.

### 3.2 Basic Search Algorithm

Before any isovalue queries, the tree infrastructure is read from disk and recreated in memory. Queries are then accepted in the form $(timestep, isovalue)$. The algorithm initially fetches the root node of the octree corresponding to $timestep$ from disk. If the extreme values stored in the root node span $isovalue$, the algorithm next fetches all children of the root node from disk. This process, shown in Fig. 2, repeats recursively until reaching the leaf nodes. If the extreme values in a leaf node span $isovalue$, the algorithm computes the disk blocks containing data points needed by

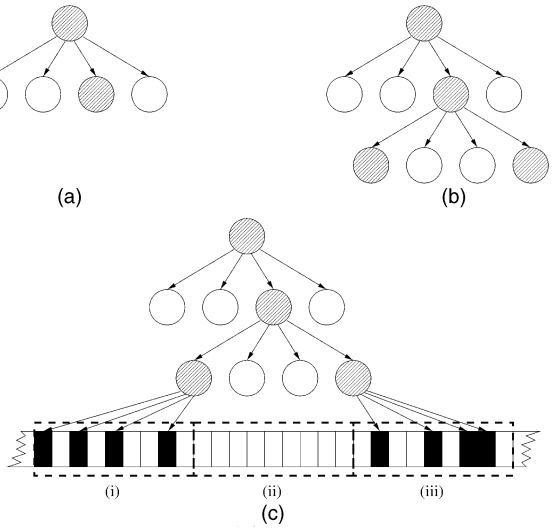


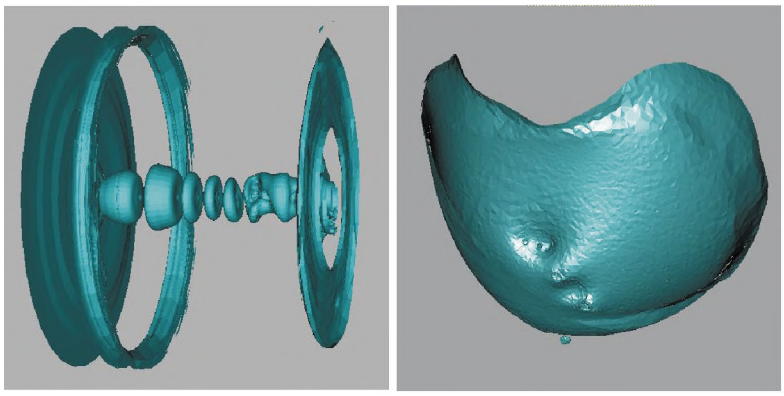

Fig. 2. The T-BON recursively brings the children of nodes that span the isovalue (indicated by shading) into memory (a, b). At the leaf level, the algorithm finds the data points needed for the isosurface (black blocks), then reads blocks containing such points (c(i), c(iii)) from disk, skipping blocks that contain only unnecessary data (c(ii)). A second pass through the tree constructs the isosurface using only the data in memory.

that leaf and inserts those blocks into a list. Once the algorithm has been brought into memory all nodes required to construct the current isosurface, it traverses the block list and reads the required data blocks sequentially from disk. This block read capitalizes on the fact that I/O systems optimize for such large-scale transfers. Some extraneous data transfers may occur, as the algorithm may not require all points in the block, but randomly accessing the data file to read strictly the required points would incur large penalties as the disk head moves between tracks and waits for the data to rotate into the proper position. Once all the required data points reside in memory, the algorithm traverses the tree a second time to construct the isosurface.

Since the T-BON data structure does not exploit temporal coherence, changing $timestep$ requires the algorithm to repeat the above process for the new isosurface query. If the user performs two sequential queries to the same timestep, the process changes to avoid rereading identical data. The T-BON data structure maintains two lists, identifying nodes and disk blocks currently in memory. By referencing these lists, the algorithm only needs to transfer differential nodes and data blocks from disk. Purging these lists when $timestep$ changes invalidates all data in memory, causing the algorithm to revert to its default behavior of reading all required nodes and data from disk. Interpolation and triangle construction times dominate the additional list processing and incremental I/O, so execution time generally equates with performing a search with the tree and data already in memory.

### 3.3 Bricking

Although the basic T-BON algorithm exploits the I/O system design by reading blocks of data, it demonstrates poor I/O performance when transferring individual nodes. To circumvent this bottleneck, the algorithm can transfer a number of nodes at once. The T-BON method packs nodes into disk blocks in order to read a number of nodes at once.



(a) (b)

Fig. 3. Isosurfaces from curvilinear and unstructured data sets. (a) Shows an isosurface from the Impinging Jet computational fluid dynamics simulation using a circular curvilinear grid. (b) Depicts an isosurface from an electrical simulation of a human torso, performed on an unstructured grid.

This strategy, called node bricking, was used by Chiang et al. [8] in their out-of-core algorithm to achieve better I/O performance. They pack $B$ binary tree nodes into one disk block-sized node with branching factor (maximum number of children) of order $\mathcal{O}(B)$. They compose these block-sized nodes into a meta-tree, which the algorithm traverses to extract an isosurface. The T-BON algorithm transfers bricks of nodes from disk during the first pass tree traversal, but simplifies the algorithm in [8] by not constructing a second tree with these meta-nodes. By maintaining and traversing the original tree, the T-BON algorithm minimizes the number of extraneous nodes read. Additionally, traversal time represents a minute fraction of total execution time, so traversing the original tree, as opposed to a meta-tree, effectively incurs no penalty.

While block reads optimize for I/O system performance, the octree traversal's cache performance may suffer because of the data layout. The octree subdivides the spatial volume in three dimensions, in contrast to the one-dimensional data stream obtained from reading blocks. Reordering the data to optimize for octree traversal can improve both memory and I/O performance. Cox and Ellsworth [23] show that cubed storage, which provides better locality of reference than one-dimensional flat storage, greatly improves out-of-core visualization performance. The T-BON algorithm utilizes this technique, also called data bricking, by implementing the formula for cubed storage given in Sakas et al. [24] and used by Parker et al. [25]. As in the node bricking technique, the T-BON data structure does not use the meta-cell method as presented in [25], again simplifying the calculations and minimizing extraneous data transfers. For some data sets, especially where isosurfaces lie against the grain of the array order, cubed storage may allow the T-BON algorithm to transfer fewer bricks from disk. In such data sets, a single brick of cubed data would contain more required data points than a single flat block.

### 3.4 Additional Grid Types

The Branch-On-Need Octree design relies on the regularity of the underlying grid. However, many simulations and measurements use nonrectilinear grids to create data sets. To extract isosurfaces from a larger number of data sets, the T-BON data structure must take into account these different types of grids. Fig. 3 shows sample isosurfaces from two simulation data sets that use different grid types. Fig. 3a depicts an isosurface from the Impinging Jet curvilinear data set. Curvilinear data sets strongly resemble rectilinear grids—they define vertex positions explicitly in space, as opposed to the implicit positions in rectilinear grids. Additionally, many curvilinear data sets contain multiple "zones," or grids, any of which may contain portions of the isosurface. The T-BON algorithm handles these grids in much the same way as rectilinear grids. The data structure stores vertex positions along with the tree structure and reads these points from disk before execution begins. Interpolation calculations reference these points rather then computing them from implicit points.

Fig. 3b shows an isosurface from an electrical simulation of a human torso. This simulation uses an unstructured grid, where tetrahedral cells replace the hexahedral cells of a regular grid. The lack of spatial hierarchy in these cells makes using a BONO-based algorithm difficult. A collection of hexahedral cells can combine to form a larger hexahedron, but tetrahedral cells may not share this property. Parker et al. [26] construct a spatial hierarchy over unstructured data by computing a maximum resolution based on the number of tetrahedra in the volume. They rectilinearly subdivide the volume to this resolution, where each leaf node maintains a linked list of the tetrahedra it contains. The T-BON algorithm takes advantage of its inherent octree structure to adaptively subdivide the volume, producing fewer nodes than a full subdivision in regions of sparse tetrahedra. Fig. 4 demonstrates this strategy in two dimensions. By producing fewer nodes, this method uses less memory and disk space.

The adaptive subdivision algorithm preprocesses the cell data using a modified version of the algorithm presented by Parker et al. [26]. This preprocess fully subdivides the volume to a certain resolution, then performs intersection tests on each tetrahedral cell to determine which subvolumes contain portions of that cell. Each subvolume maintains a list of all cells contained by that subvolume. Finally, the preprocess concatenates these lists into a single

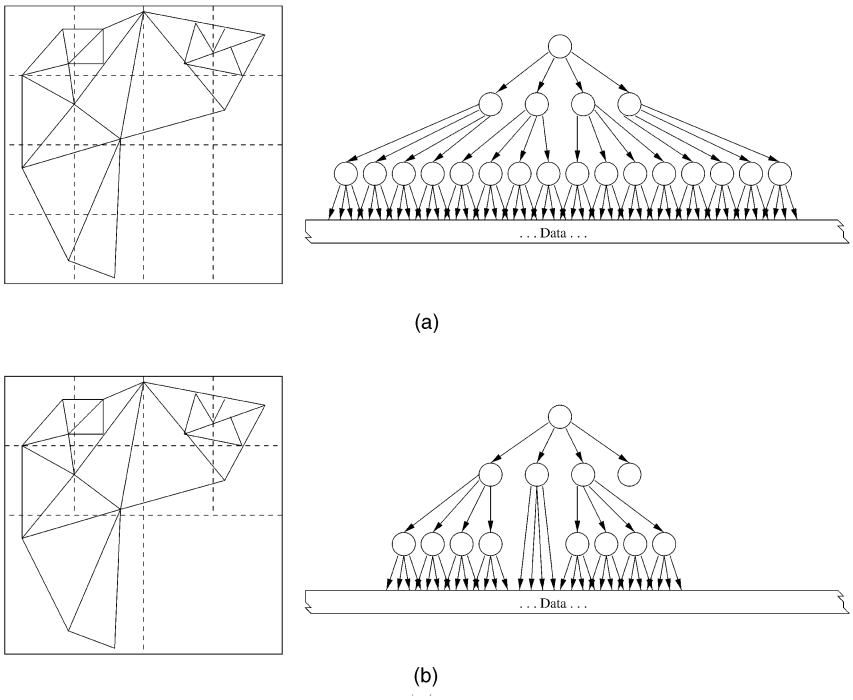



Fig. 4. Comparing adaptive refinement to full-resolution subdivision. (a) Shows the full-resolution scheme of Parker et al. [26]. Using the octree structure of the T-BON to perform adaptive refinement, as shown in (b), results in fewer nodes in sparse and empty regions.

array, which it writes to disk. A second preprocessing program constructs a BONO over the subvolumes. The T-BON preprocessing algorithm then traverses the BONO using a user-specified threshold. If all children of a node contain fewer cells than this threshold, the children merge their linked lists into a single cell list, removing duplicate cells if necessary. The T-BON then collapses the children, making their parent a leaf node and assigning it the merged list. This process continues in a bottom-up manner until all nodes contain a number of cells greater than or equal to the threshold value. Finally, the process writes the condensed tree and lists to disk, using less storage space than the full-resolution decomposition.

### 3.5 Out-of-Core Algorithms

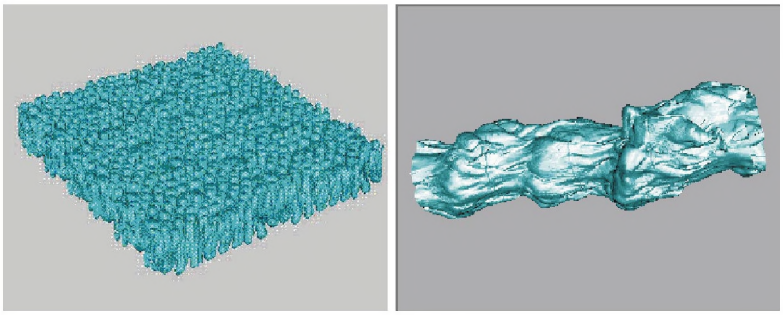
Systems with limited memory can perform isosurface extraction using an out-of-core technique. These algorithms swap data between disk and memory more efficiently than the virtual memory system, allowing the user to visualize data sets larger than main memory. We examine two related out-of-core algorithms based on the T-BON data structure. Both make use of dynamic data structures that resemble a hardware cache. These structures can store a user-defined number of node or data bricks during traversal. A cache "line" in this structure contains memory for a single brick, a "valid" bit, and a number indicating the time of last access to the brick. The out-of-core algorithms must clear all valid bits when the time value of the isosurface query changes since the T-BON does not utilize temporal coherence. Both algorithms use the time of last access and a least recently used (LRU) scheme to determine which brick to replace when the cache fills.

The first and most flexible out-of-core algorithm uses two of these dynamic structures, one for node bricks and one for data bricks. To use the least amount of memory, this algorithm uses a depth-first search traversal of the tree. A preprocess reorders node bricks, originally packed in a breadth-first manner, into a depth-first pattern and stores the reordered bricks to disk. An iterative depth-first traversal uses a stack to keep track of which node bricks reside in memory. The data brick cache must contain space for a minimum of eight bricks since a cell along a brick boundary may reference points in eight neighboring bricks. To avoid rereading bricks already in memory, the list of data bricks updates the LRU fields of any resident brick required by the current node before selecting a brick to replace.

The second out-of-core algorithm performs better than the algorithm above, at the expense of a higher memory requirement. This method initially allocates space for one full BONO and reads nodes from disk as in the basic in-core algorithm. However, this technique uses a dynamic cache structure for the data bricks. Since the memory required to store the data dominates the memory required for the tree, this technique still presents substantial storage savings. The user can still tune the memory usage by altering the size of the dynamic data structure, although to a lesser extent than in the first algorithm. This methods offers higher performance because it accesses the node and data files sequentially, as opposed to the random access required by the first method. This capitalizes on the I/O system design, which optimizes for sequential access, and therefore minimizes latency due to disk head movement.

## 4 Results

The T-BON algorithm was tested using a number of computational fluid dynamics (CFD) simulation data sets.



(a) (b)

Fig. 5. Isosurfaces from test data sets. (a) Depicts an isosurface from the RAGE simulation, showing the bubbles formed by Rayleigh-Taylor instability. (b) Shows an isosurface from the Jet Shockwave simulation, demonstrating the instabilities in a supersonic jet.

The original Rage data set contains $512 \times 512 \times 512$ points per time step and represents a simulation of the Rayleigh-Taylor hydrodynamic instability, where two fluids of different densities mix. Each data point contains a single floating-point value representing the density at that grid point. Fig. 5a shows a sample isosurface from this data set. The label "Rage512" designates the original data set, while "Rage128" represents a subsampled version of the data, containing $128 \times 128 \times 128$ points per time step, or a sampling of every fourth point in each dimension from the Rage512 data set. The jet shockwave data set, labeled "Jet256," contains $256 \times 256 \times 256$ points per time step, each corresponding to a one-byte unsigned character value. This data set simulates the Kelvin-Helmholz instability in a supersonic jet. Fig. 5b displays a sample isosurface from this data set. The jet shockwave data set and all versions of the Rage data set use rectilinear grids to implicitly denote positions of the data points in space.

The Impinging Jet data set represents a jet simulation using three curvilinear grids, totaling 877,568 data points per time step. The Torso data set represents an electrical simulation of a human torso using an unstructured grid. This data set contains 168,706 points and 1,081,280 tetrahedral cells.

Results in this section represent two types of experiments. The first tests the dynamic behavior of the T-BON data structure by holding the isovalue constant and varying the time value. This corresponds to observing the change in an isosurface of interest over time. The T-BON optimizes for this behavior. The second experiment type holds the time value constant and varies the isovalue. This corresponds to searching for a surface of interest and effectively tests the behavior of the algorithm on a static data set. All experiments ran on a single dedicated processor of an SGI Origin 2000 (32 250MHz R10000 processors) with 8GB of memory.

Many of the results below display values for "speedup." These values use a pure BONO approach for comparison. Experimental comparisons [22] show that the branch-on-need octree is the best known geometric acceleration technique for three-dimensional static data sets. Since it demonstrates good performance and memory behavior and provides the basis for the T-BON algorithm, this technique represents the fairest comparison for time-varying data sets. The three-dimensional BONO algorithm performs isosurface extraction in time-varying fields by reading all nodes and all data into memory whenever the time value in the isosurface query changes. When the three-dimensional BONO method receives two successive isosurface queries with the same time value but different isovalues, it does not reread the data for the given time step, but simply begins isosurface extraction using the data resident in memory. In contrast, the T-BON algorithm resolves two queries with the same time value by incrementally reading the additional nodes and data needed to construct the isosurface.

### 4.1 Bricking Results

Table 1 shows results from the Rage512 data set using the basic T-BON algorithm, the algorithm using node bricking, and the algorithm using both node and data bricking. This table shows the best, average, and worst case per query speedups, averaged over eight representative isovalues and five time steps. Times for the basic algorithm (the "No Bricking" section) are not identical to those given in [9]—a cleaner implementation has improved the average and

TABLE 1
Results for a Dynamic Data Set—Rage512

| No Bricking | | | |
|---|---|---|---|
| | Min | Average | Max |
| speedup | 3.51 | 3.80 | 4.43 |
| # triangles | 4,822,484 | 4,342,715 | 5,062,684 |
| % nodes read | 7.32 | 6.70 | 7.59 |
| % data read | 12.09 | 11.13 | 12.09 |
| **Node Bricking** | | | |
| | Min | Average | Max |
| speedup | 3.32 | 6.14 | 7.21 |
| # triangles | 4,777,764 | 4,342,715 | 5,003,372 |
| % nodes read | 32.63 | 31.35 | 32.95 |
| % data read | 11.72 | 11.13 | 12.50 |
| **Node and Data Bricking** | | | |
| | Min | Average | Max |
| speedup | 4.85 | 6.62 | 7.56 |
| # triangles | 4,777,764 | 4,342,715 | 5,003,372 |
| % nodes read | 32.63 | 31.15 | 32.95 |
| % data read | 13.91 | 13.37 | 14.61 |

TABLE 2
Results for a Dynamic Data Set—Jet256

| No Bricking | | | |
|---|---|---|---|
| | Min | Average | Max |
| speedup | 3.41 | 8.26 | 10.85 |
| # triangles | 365,672 | 179,785 | 104,636 |
| % nodes read | 4.40 | 2.32 | 1.50 |
| % data read | 21.12 | 9.92 | 4.05 |
| Node Bricking | | | |
| | Min | Average | Max |
| speedup | 7.80 | 23.06 | 32.70 |
| # triangles | 368,078 | 179,785 | 513,484 |
| % nodes read | 30.93 | 17.53 | 38.45 |
| % data read | 20.43 | 9.92 | 26.22 |
| Node and Data Bricking | | | |
| | Min | Average | Max |
| speedup | 7.98 | 23.82 | 34.25 |
| # triangles | 365,672 | 179,785 | 513,484 |
| % nodes read | 30.93 | 17.53 | 38.45 |
| % data read | 9.42 | 5.47 | 12.26 |

TABLE 4
Results for a Static Data Set—Jet256

| No Bricking | | | |
|---|---|---|---|
| | Min | Average | Max |
| speedup | 0.58 | 1.43 | 10.12 |
| # triangles | 320,236 | 179,785 | 104,636 |
| % nodes read | 0.63 | 0.49 | 0.063 |
| % data read | 0.00 | 1.35 | 0.049 |
| Node Bricking | | | |
| | Min | Average | Max |
| speedup | 0.95 | 3.08 | 28.92 |
| # triangles | 106,890 | 179,785 | 102,000 |
| % nodes read | 0.00 | 3.86 | 0.00 |
| % data read | 0.00 | 1.45 | 0.00 |
| Node and Data Bricking | | | |
| | Min | Average | Max |
| speedup | 0.90 | 3.02 | 28.92 |
| # triangles | 107,614 | 179,785 | 105,420 |
| % nodes read | 0.043 | 2.58 | 0.085 |
| % data read | 0.16 | 1.20 | 0.34 |

worst case speedups for both data sets. Table 2 shows results for similar experiments on the Jet256 data set, averaged over eight representative isovalues and ten time steps. Tables 1 and 2 also show the percentages of nodes and data read in the best, average, and worst cases, along with the number of triangles constructed in each case.

These tables show large speedups over the BONO algorithm. The T-BON's lower I/O latency produces a factor of 3.8 improvement (average case) in the Rage512 data set and an 8.3 times speedup for the Jet256 data set. The BONO algorithm must read in large amounts of unnecessary data before constructing an isosurface, while, even in the worst case of minimum speedup, the T-BON reads in less than 10 percent of the nodes and less than a third of the data for both data sets.

TABLE 3
Results for a Static Data Set—Rage512

| No Bricking | | | |
|---|---|---|---|
| | Min | Average | Max |
| speedup | 0.80 | 1.07 | 3.73 |
| # triangles | 4,017,388 | 4,342,715 | 3,992,268 |
| % nodes read | 6.40 | 1.00 | 0.18 |
| % data read | 10.86 | 1.54 | 0.00 |
| Node Bricking | | | |
| | Min | Average | Max |
| speedup | 0.97 | 1.31 | 4.64 |
| # triangles | 5,062,684 | 4,342,715 | 3,992,268 |
| % nodes read | 24.56 | 3.00 | 0.15 |
| % data read | 12.09 | 1.86 | 0.00 |
| Node and Data Bricking | | | |
| | Min | Average | Max |
| speedup | 1.07 | 1.56 | 6.18 |
| # triangles | 5,062,684 | 4,342,715 | 3,992,268 |
| % nodes read | 32.90 | 4.08 | 0.064 |
| % data read | 14.19 | 2.55 | 0.078 |

By better exploiting the design of the I/O system, bricking further improves performance. Bricking only the nodes produces a large increase in speedup—average case speedups increase by factors of 1.6 and 2.8 for the Rage512 and Jet256 data sets, respectively. In the worst case, node bricking may slow the algorithm because of the extra computation (shown in Table 1 under "Node Bricking"). Bricking both nodes and data results in further small improvements and can lead to large improvements in the worst cases. For example, the worst case speedup for the Rage512 data set improves by 38 percent over the basic algorithm by using node and data bricking.

Although the T-BON data structure optimizes for dynamic isovalue queries, it must also perform well for static queries to be useful. Tables 3 and 4 show the results for this static behavior for the Rage512 and Jet256 data sets. These experiments use the same isovalues and time steps as those in Tables 1 and 2, but hold the time value constant and vary the isovalue. These results show that, although the basic T-BON can perform much better than the pure BONO (best case speedups of 3.73 and 10.12), in the worst case it performs half as well. While the BONO reads all nodes and data from disk once per time step, the basic T-BON algorithm must continually access the disk as the isovalue changes, incrementally bringing additional nodes and data into memory. Eventually, all nodes and data will reside in memory and the performance of the T-BON algorithm will converge to that of the BONO. This demonstrates the major shortcoming of the previously published T-BON algorithm. By not fully exploiting the I/O system, performance on static data sets can suffer.

Tables 3 and 4 show that node bricking, which demonstrates much better I/O behavior, again results in

TABLE 5
Results for the Curvilinear Impinging Jet Data Set

| Dynamic data set | | | |
|---|---|---|---|
| | Min | Average | Max |
| speedup | 1.36 | 3.05 | 4.71 |
| # triangles | 136,100 | 109,277 | 51,601 |
| % nodes read | 60.86 | 51.06 | 29.67 |
| % data read | 61.86 | 52.08 | 30.79 |
| Static data set | | | |
| speedup | 1.05 | 1.32 | 1.91 |
| # triangles | 135,673 | 109,277 | 54,348 |
| % nodes read | 0.072 | 0.083 | 0.0021 |
| % data read | 11.63 | 12.51 | 1.85 |

TABLE 6
Adaptive Subdivision Results for the Torso Data Set

| $Threshold$ | # nodes | # subvolumes | Execution time |
|---|---|---|---|
| 0 | 206,456 | 180,000 | 7.48 |
| 1 | 157,616 | 137,637 | 7.12 |
| 2 | 156,616 | 136,765 | 7.23 |
| 3 | 155,896 | 136,138 | 7.38 |
| 4 | 154,908 | 135,278 | 7.16 |
| 5 | 153,868 | 134,373 | 7.21 |
| 10 | 149,584 | 130,638 | 7.36 |
| 20 | 139,920 | 122,222 | 7.22 |
| 30 | 123,906 | 108,244 | 7.28 |
| 50 | 102,398 | 89,469 | 7.37 |
| 100 | 56,068 | 49,011 | 7.50 |

large performance improvements. As in the dynamic case, these tables show that the algorithm reads more nodes from disk since not all nodes in a brick span the isovalue. However, this block read improves performance significantly over the base algorithm since it better exploits the I/O system design. Node bricking improves average performance by a factor of 1.2 in the Rage512 data set and by a factor of 2.2 in the Jet256 data set. Bricking both nodes and data causes the T-BON to perform better in the worst case (speedup factor of 1.07) than the BONO algorithm for the Rage512 data set, but the addition of more computation decreases the performance by 5.6 percent in the Jet256 data set. However, the T-BON algorithm performs only around 5 percent worse than the BONO when using node bricking. This represents a significant improvement over the 42 percent difference of the basic algorithm and shows that, with the addition of bricking, the T-BON algorithm almost never performs worse than the BONO implementation.

### 4.2 Curvilinear and Unstructured Grid Results

Experiments performed on the curvilinear T-BON implementation used the Impinging Jet data set with 11 isovalues and 10 time steps. Table 5 shows the overall behavior for both the static and dynamic cases using node and data bricking. The isosurfaces in this data set occupy large portions of the volume, requiring the algorithm to read approximately half the data for the average query. This accounts for the lower speedups compared to the same technique in the Rage512 and Jet256 data sets. However, node and data bricking again ensure that the T-BON algorithm never performs worse than the curvilinear BONO technique.

The Torso data set contains only a single time step, making a fair comparative performance experiment impossible. However, experiments that test the adaptive subdivision algorithm show the efficacy of this technique. The preprocess initially subdivides the volume to a resolution of $150 \times 64 \times 150$, then adaptively collapses nodes until all nodes contain at least $Threshold$ cells. A value of zero for $Threshold$ causes the algorithm to use the full resolution subdivision. Table 6 shows total execution times for the single time step using 35 isovalues. This table shows that the largest improvement occurs when $Threshold$ changes from 0 to 1. This corresponds to collapsing empty regions of the volume, but maintaining those portions of the tree that contain at least one cell. As $Threshold$ increases, performance slowly degrades as the linked lists of cells at each node increase in size, causing the algorithm to visit more cells that do not contain portions of the isosurface. Storage requirements for the tree shrink as $Threshold$ increases, a manifestation of the storage/performance trade-off inherent in isosurface acceleration techniques. Simply collapsing the empty regions by setting $Threshold$ to 1 provides the best combination of storage space and performance.

### 4.3 Out-of-Core Results

Experiments on the more flexible out-of-core algorithm described in Section 3.5 used the dynamic Jet256 and Rage128 data sets. The tests queried the T-BON data structure with four isovalues and five time steps for each data set. Results from varying the sizes of the dynamic data structures are shown in Table 7. By using dynamic data structures, this algorithm can perform isosurface extraction on data sets much larger than its memory footprint. By using less than 1 percent of the memory required to store the data set, the T-BON-based out-of-core algorithm can extract isosurfaces in a reasonable amount of time. For better performance using this design, the size of the data brick cache must be kept small because the triangle construction method accesses it so frequently. The cache tags are searched linearly to find the correct brick, so a large cache requires a linearly higher average search time.

TABLE 7
Results from the First Out-of-Core Algorithm

| Jet256 (Data size: 16.8MB) | | |
|---|---|---|
| | Memory Used (MB) | Execution Time (s) |
| T-BON | 47 | 0.25 |
| Out-of-Core | 0.12 - 8.52 | 7.07 - 8.55 |
| Rage128 (Data size: 8.4MB) | | |
| | Memory Used (MB) | Execution Time (s) |
| T-BON | 16 | 0.51 |
| Out-of-Core | 0.15 - 8.4 | 1.84 - 4.90 |

TABLE 8
Results from the Second Out-of-Core Algorithm

| Jet256 (Data size: 16.8MB) | | |
|---|---|---|
| | Memory Used (MB) | Execution Time (s) |
| T-BON | 47 | 0.25 |
| Out-of-Core | 19.5 | 0.41 |
| Rage128 (Data size: 8.4MB) | | |
| | Memory Used (MB) | Execution Time (s) |
| T-BON | 16 | 0.51 |
| Out-of-Core | 5.2 | 0.78 |

Experiments on the second out-of-core algorithm described in Section 3.5 used the same data sets and queries as the first. The T-BON algorithm reads node bricks into a static structure and reads data bricks into a dynamic cache structure. Table 8 shows results for this algorithm using the smallest data brick cache size. This algorithm performs better because of better I/O utilization, decreasing the memory requirement by 59 percent for the Jet256 data set while increasing the execution time by only 64 percent, as opposed to the much higher execution time of the first algorithm. For the Rage128 data set, memory usage is decreased by 68 percent while execution time increases by only 58 percent.

These two algorithms provide limited memory systems with the ability to perform isosurface extraction on large, time-varying data sets with reasonable performance. By adjusting the parameters of these two algorithms, users can choose a suitable compromise between storage space and performance based on system configurations.

## 5 Conclusions and Future Work

The Temporal Branch-on-Need Octree (T-BON) accelerates isosurface extraction in time-varying fields by maintaining a low memory profile and minimizing the effects of the I/O bottleneck. By transferring only those portions of the tree and data necessary to construct the current isosurface, the T-BON data structure delivers high performance and significant speedups over a static algorithm.

The T-BON algorithm quantifies the acceleration possible by using a series of three-dimensional data structures and selective disk transfers, without exploiting the temporal locality of the isosurfaces. It can perform isosurface extraction on most of the grid types in common usage, including rectilinear, curvilinear, and unstructured. The T-BON algorithm can also use out-of-core techniques to perform isosurface extraction on systems with limited memory.

Several directions for future work exist. Performing comparisons between the T-BON algorithm and other time-varying algorithms, such as the Temporal Hierarchical Index Tree [21] and the four-dimensional field technique proposed by Weigle and Banks [20], would quantify the benefits of utilizing temporal coherence versus using low memory overhead and selective disk transfers. Combining the best attributes of the T-BON and temporal coherence algorithms might improve performance further for dynamic data sets. The T-BON data structure does not currently utilize temporal coherence, but does not preclude its inclusion. For example, a combination of the T-BON and Temporal Hierarchical Index Tree data structures could capture temporal coherence while providing lower memory and I/O overhead. Instead of constructing an ISSUE data structure at each node of the tree, this algorithm could substitute a T-BON data structure, allowing it to reduce the impact of the I/O bottleneck.

The technique of selective disk transfers could apply to geometric data structures other than the branch-on-need octree. For example, Parker et al. [25] use a shallow hierarchy, where each node has a large number of children (much more than eight). A T-BON implementation using this data structure could improve performance for some data sets since it could more easily ignore large areas of the volume that do not intersect the isosurface.

Investigating additional out-of-core techniques could result in better performance for low-memory systems. Incorporating the meta-brick strategy of Chiang et al. [7], [8] and developing better dynamic data structures could benefit this technique.

## Acknowledgments

This work was supported in part by the C-SAFE DOE ASCI Alliance Center, the DOE Advanced Visualization Technology Center (AVTC), and a grant from Lawrence Livermore National Laboratory. The Rage data set is courtesy of Robert Weaver (Los Alamos National Laboratory). The Jet Shockwave data set was obtained from the Advanced Visualization Technology Center data repository at Argonne National Laboratory (http://avtc-data.mcs.anl.gov/). The Impinging Jet data set was obtained from the NASA archive (http://www.nas.nasa.gov/Software/DataSets/) and the Torso data set is courtesy of Ruth Klepfer from the University of Utah.

## References

[1] W.E. Lorensen, "Marching through the Visible Man," *Proc. Visualization 1995*, pp. 368-373, Oct. 1995.
[2] U. Tiede, T. Schiemann, and K.H. Höhne, "High Quality Rendering of Attributed Volume Data," *Proc. Visualization 1998*, pp. 255-262, Oct. 1998.
[3] J.M. Favre, "Towards Efficient Visualization Support for Single-Block and Multi-Block Datasets," *Proc. Visualization 1997*, pp. 423-428, Oct. 1997.
[4] P.D. Heermann, "Production Visualization for the ASCI One Teraflops Machine," *Proc. Visualization 1998*, pp. 459-462, Oct. 1998.
[5] M. Lanzagorta, M.V. Kral, J.E. Swan II, G. Spanos, R. Rosenberg, and E. Kuo, "Three-Dimensional Visualization of Microstructures," *Proc. Visualization 1998*, pp. 487-490, Oct. 1998.
[6] C.R.F. Monks, P.J. Crossno, G. Davidson, C. Pavlakos, A. Kupfer, C. Silva, and B. Wylie, "Three Dimensional Visualization of Proteins in Cellular Interactions," *Proc. Visualization 1996*, pp. 363-366, Oct. 1996.
[7] Y. Chiang and C.T. Silva, "I/O Optimal Isosurface Extraction," *Proc. Visualization 1997*, pp. 293-300, Oct. 1997.
[8] Y. Chiang, C.T. Silva, and W.J. Schroeder, "Interactive Out-of-Core Isosurface Extraction," *Proc. Visualization 1998*, pp. 167-174, Oct. 1998.
[9] P. Sutton and C.D. Hansen, "Isosurface Extraction in Time-Varying Fields Using a Temporal Branch-on-Need Tree (T-BON)," *Proc. Visualization 1999*, pp. 147-153, Oct. 1999.

[10] W.E. Lorensen and H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics*, vol. 21, pp. 163-169, July 1987.
[11] G. Wyvill, C. McPheeters, and B. Wyvill, "Data Structure for Soft Objects," *The Visual Computer*, vol. 2, no. 4, pp. 227-234, 1986.
[12] J. Wilhelms and A. van Gelder, "Octrees for Faster Isosurface Generation," *ACM Trans. Graphics*, vol. 11, no. 3, pp. 201-227, July 1992.
[13] Y. Livnat, H. Shen, and C.R. Johnson, "A Near Optimal Isosurface Extraction Algorithm Using the Span Space," *IEEE Trans. Visualization and Computer Graphics*, vol. 2, no. 1, pp. 73-84, Mar. 1996.
[14] H. Shen, C.D. Hansen, Y. Livnat, and C.R. Johnson, "Isosurfacing in Span Space with Utmost Efficiency (Issue)," *Proc. Visualization 1996*, pp. 287-294, Oct. 1996.
[15] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno, "Speeding Up Isosurface Extraction Using Interval Trees," *IEEE Trans. Visualization and Computer Graphics*, vol. 3, no. 2, pp. 158-170, Apr.-June 1997.
[16] T. Itoh and K. Koyamada, "Automatic Isosurface Propagation Using an Extrema Graph and Sorted Boundary Cell Lists," *IEEE Trans. Visualization and Computer Graphics*, vol. 1, no. 4, pp. 319-327, Dec. 1995.
[17] T. Itoh, Y. Yamaguchi, and K. Koyamada, "Volume Thinning for Automatic Isosurface Propagation," *Proc. Visualization 1996*, pp. 303-310, Oct. 1996.
[18] C.L. Bajaj, V. Pascucci, and D.R. Schikore, "Fast Isocontouring for Improved Interactivity," *Proc. 1996 Symp. Volume Visualization*, pp. 39-46, Oct. 1996.
[19] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore, "Contour Trees and Small Seed Sets for Isosurface Traversal," *Proc. 13th Ann. Symp. Computational Geometry*, pp. 212-220, June 1997.
[20] C. Weigle and D.C. Banks, "Extracting Iso-Valued Features in 4-Dimensional Scalar Fields," *Proc. 1998 Symp. Volume Visualization*, pp. 103-110, Oct. 1998.
[21] H. Shen, "Isosurface Extraction in Time-Varying Fields Using a Temporal Hierarchical Index Tree," *Proc. Visualization 1998*, pp. 159-164, Oct. 1998.
[22] P.M. Sutton, C.D. Hansen, H. Shen, and D. Schikore, "A Case Study of Isosurface Extraction Algorithm Performance," *Proc. Joint Eurographics-IEEE TCVG Symp. Visualization*, to appear, May 2000.
[23] M. Cox and D. Ellsworth, "Application-Controlled Demand Paging for Out-of-Core Visualization," *Proc. Visualization 1997*, pp. 235-244, Oct. 1997.
[24] G. Sakas, M. Grimm, and A. Savopoulos, "Optimized Maximum Intensity Projection (MIP)," *Proc. Eurographics Rendering Workshop 1995*, June 1995.
[25] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. Sloan, "Interactive Ray Tracing for Isosurface Rendering," *Proc. Visualization 1998*, pp. 233-238, Oct. 1998.
[26] S. Parker, M. Parker, Y. Livnat, P. Sloan, C. Hansen, and P. Shirley, "Interactive Ray Tracing for Volume Visualization," *IEEE Trans. Visualization and Computer Graphics*, vol. 5, no. 3, pp. 238-250 July-Aug. 1999.

**Philip M. Sutton** recently joined the visualization group at Lawrence Livermore National Laboratory, after completing the requirements for an MS in computer science at the University of Utah. His interests include large-scale time-varying scientific visualization, parallel visualization algorithms and computer graphics. He received a BS in engineering and applied science from Caltech in 1997.

**Charles D. Hansen** received a BS in computer science from Memphis State University in 1981 and a PhD in computer science from the University of Utah in 1987. He is an associate professor of computer science at the University of Utah. From 1997 to 1999, he was a research associate professor in computer science at the University of Utah. From 1989 to 1997, he was a technical staff member in the Advanced Computing Laboratory (ACL) located at Los Alamos National Laboratory, where he formed and directed the visualization efforts in the ACL. He was a Bourse de Chateaubriand postdoctoral fellow at INRIA, Rocquencourt, France, in 1987 and 1988. His research interests include large-scale scientific visualization, parallel computer graphics algorithms, massively parallel processing, 3D shape representation, and computer vision. He is a member of the IEEE and the IEEE Computer Society.