A Near Optimal Isosurface Extraction Algorithm Using The Span Space

Yarden Livnat, Han-Wei Shen and Christopher R. Johnson

Abstract—
We present the "Near Optimal IsoSurface Extraction" (NOISE) algorithm for rapidly extracting isosurfaces from structured and unstructured grids. Using the *span space*, a new representation of the underlying domain, we develop an isosurface extraction algorithm with a worst case complexity of $O(\sqrt{n}+k)$ for the *search phase*, where n is the size of the data set and k is the number of cells intersected by the isosurface. The memory requirement is kept at $O(n)$ while the preprocessing step is $O(n \log n)$. We utilize the span space representation as a tool for comparing isosurface extraction methods on structured and unstructured grids. We also present a fast triangulation scheme for generating and displaying unstructured tetrahedral grids.

Keywords—**Isosurface extraction, unstructured grids, span space, kd-trees**

I. Introduction

Isosurface extraction is a powerful tool for investigating scalar fields within volumetric data sets. The position of an isosurface, as well as its relation to other neighboring isosurfaces, can provide clues to the underlying structure of the scalar field. In medical imaging applications, isosurfaces permit the extraction of particular anatomical structures and tissues. These isosurfaces are static in nature. A more dynamic use of isosurfaces is called for in many computational science applications, such as computational fluid dynamics and atmospheric simulations. In such applications, scientists would ideally like to dynamically investigate the scalar field in order to gain better insight into simulation results.

As scientific computation demands higher accuracy and state-of-the-art medical scanners increase in resolution, the resulting data sets for visualization expand rapidly. The sheer size of these data sets, as well as their structure, pose major obstacles for interactive investigation. While medical imaging data usually structured in nature, other scientific and engineering data sets frequently consist of geometry represented by unstructured finite element grids.

Originally, isosurface extraction methods were restricted to structured grid geometry, as such, early efforts focused on extracting a single isosurface [1] from the volumetric data set. Recently, in an effort to speed up isosurface extraction, several methods were developed that could be adapted to extraction of multiple isosurfaces from structured [2], [3] as well as from unstructured geometry [4], [5]. Nevertheless, for large data sets, existing methods do not allow for interactive investigation of the data set, especially for unstructured grids. Defining n as the number of data cells and k as the number of cells intersecting a given isosurface, most of the existing algorithms have time complexity of $O(n)$. While [2] has an improved time complexity of $O(k \log(\frac{n}{k}))$, the algorithm is only suitable for structured hexahedral grids.

The authors are with the Department of Computer Science, University of Utah, Salt Lake City, UT 84112 E-Mail: (ylivnat,hwshen,crj)@cs.utah.edu Web: http://www.cs.utah.edu/~sci/

In this paper we introduce a new view of the underlying domain. We call this new representation the *span space*. Based on this new perspective, we propose a fast and efficient, $O(\sqrt{n}+k)$, isosurface extraction algorithm for both structured and unstructured grids.

Section II investigates the underlying domain for structured and unstructured problems and the new decomposition of this domain is then proposed. The proposed Span Space is then used in section III as a common backdrop for comparing previous methods of isosurface extraction. Section IV shows how the Span Space paradigm leads to an efficient representation and fast isosurface extraction methods. In section V, we present several optimizations with respect to both memory and time requirements. A fast triangulation method for unstructured tetrahedral grid is presented in Section VI. We conclude by analyzing the results of testing the new algorithm on several science and engineering applications.

II. The Span Space

Let $\varphi : \mathbb{G} \to \mathbb{V}$ be a given field and let $\mathcal{D}$ be a sample set over φ, such that,

$$\mathcal{D} = \{d_i\} \qquad d_i \in \mathbb{D} = \mathbb{G} \times \mathbb{V}$$

where $\mathbb{G} \subseteq \mathbb{R}^p$ is a geometric space and $\mathbb{V} \subseteq \mathbb{R}^q$ is the associated value space, for some $p, q \in \mathbb{Z}^+$. Also, let $d = |\mathcal{D}|$ be the size of the data set.

Definition 1 (Isosurface Extraction) Given a set of samples $\mathcal{D}$ over a field $\varphi : \mathbb{G} \to \mathbb{V}$, and given a single value $v \in \mathbb{V}$, find,

$$S = \{g_i\} \quad g_i \in \mathbb{G} \quad \text{such that,} \quad \varphi(g_i) = v. \tag{1}$$

Note that S, the *isosurface,* need not be topologically simple.

Approximating an isosurface, S, as a global solution to Eq. 1 can be a difficult task because of the sheer size, d, of a typical science or engineering data set.

Data is often generated from 3D images or as solutions to numerical approximation techniques, such as from finite difference or finite element methods. These methods naturally decompose the geometric space, $\mathbb{G}$, into a set of polyhedral cells, C, where the data points define the vertices. Rather than finding a global solution one can seek a local approximation within each cell. Hence, isosurface extraction becomes a two-stage process: Locating the cells that intersect the isosurface and then, locally, approximating the isosurface inside each such cell. We focus our attention on the problem of finding those cells that intersect an isosurface of a specified isovalue.

On structured grids, the position of a cell can be represented in the geometric space $\mathbb{G}$. Because this representation does not require explicit adjacency information between cells, isosurface extraction methods on structured grids conduct searches over the geometric space, $\mathbb{G}$. The problem as stated by these methods is defined as follows:

Approach 1 (Geometric Search) Given a point $v \in \mathbb{V}$ and given a set C of cells in $\mathbb{G}$ space where each cell is associated with a set of values $\{v_j\} \in \mathbb{V}$, find the subset of C which an isosurface, of value v, intersects.

Efficient isosurface extraction for unstructured grids is more difficult, as no explicit order, i.e. position and shape, is imposed on the cells, only an implicit one that is difficult to utilize. Methods designed to work in this domain have to use additional explicit information or revert to a search over the value space, $\mathbb{V}$. The advantage of the latter approach is that one needs only to examine the minimum and maximum values of a cell to determine if an isosurface intersects that cell. Hence, the dimensionality of the problem reduces to two for scalar fields.

Current methods for isosurface extraction over unstructured grids, as well as some for structured grids, view the isosurface extraction problem in the following way:

Approach 2 (Interval Search) Given a point $v \in \mathbb{V}$ and given a set of cells represented as intervals,

$$I = \{[a_i, b_i]\} \qquad \text{such that,} \quad a_i, b_i \in \mathbb{V}$$

find the subset I_s such that,

$$I_s \subseteq I \qquad \text{and,} \quad a_i \leq v \leq b_i \quad \forall [a_i, b_i] \in I_s,$$

where a norm should be used when the dimensionality of $\mathbb{V}$ is greater than one.

Posing the search problem over intervals introduces some difficulties. If the intervals are of the same length or are mutually exclusive they can be organized in an efficient way suitable for quick queries. However, it is much less obvious how to organize an arbitrary set of intervals. Indeed, what distinguishes these methods from one another is the way they *organize* the intervals rather than how they perform searches.

A key point is that the minimum and maximum values are given over the same dimension. More formally, the minimum and maximum values are represented over a basis that includes only one unit vector. This degenerated basis is the cause for the above difficulties. We should be able to obtain a simpler representation if we use a basis that includes two unit vectors, one for the min value and one for the max value. Better still, the maximum separation between the representation of the min and max values will occur when these two unit vectors are perpendicular to each other. We are, therefore, led to a new representation, *a point in a plane*, using the natural coordinate system to represent the minimum and maximum values.

The method proposed in this paper addresses the problem of isosurface generation over unstructured grids and searches over the value space. Our approach, nevertheless, is not to view the problem as a search over intervals in $\mathbb{V}$ but rather as a search over points in $\mathbb{V}^2$. We start with an augmented definition of the search space.

Definition 2 (The Span Space) Let C be a given set of cells, define a set of points $P = \{p_i\}$ over $\mathbb{V}^2$ such that,

$$\forall c_i \in C \qquad \text{associate,} \quad p_i = (a_i, b_i)$$

where,

$$a_i = \min_j \{v_j\}_i \quad \text{and} \quad b_i = \max_j \{v_j\}_i$$

and $\{v_j\}_i$ are the values of the vertices of cell i.

Though conceptually not much different than the interval space, the span space will, nevertheless, lead to a simple and near optimal search algorithm. In addition, the span space will enable us to clarify the differences and commonalities between previous interval approaches.

The benefit of using the the span space is that points in 2D exhibit no explicit relations between themselves, while intervals tend to be viewed as stacked on top of each other, so that overlapping intervals exhibit merely coincidental links. Points, do not exhibit such arbitrary ties and in this respect lend themselves to many different organizations. However, as we shall show later, previous methods grouped these points in very similar ways, because they looked at them from an interval perspective.

Using our augmented definition, the isosurface extraction problem can be stated as,

Approach 3 (The Span Search) Given a set of cells, C, and its associated set of points, P, in the span space, and given a value $v \in \mathbb{V}$, find the subset $P_s \subseteq P$, such that

$$\forall (x_i, y_i) \in P_s \quad x_i < v \quad y_i > v.$$

We note that $\forall (x_i, y_i) \in P_s, \quad x_i < y_i$ and thus the associated points will lie above the line $y_i = x_i$. A geometric perspective of the span search is given in Fig. 1.

III. Previous Work

We now examine previous approaches to the problem of isosurface generation.

A. Geometric Space Decomposition

Originally, only structured grids were available as an underlying geometry. Structured grids impose order on the given cell set. This fact helps to keep the geometric complexity of the entire cell set in $\mathbb{G}$. By utilizing this order, methods based on the geometry of the data set could take advantage of the coherence between adjacent cells.

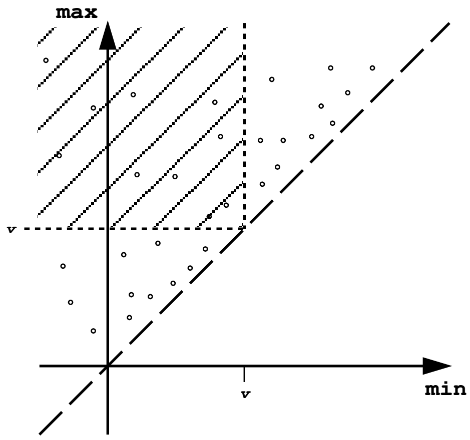

Fig. 1. Search over the span space. A data cell is represented by a point based upon the minimum and maximum values at the vertices of the cell. The points in the shaded area represent the cells that intersect the isovalue v.

A.1 Marching Cubes

Perhaps the most well known isosurface extraction method to achieve high resolution results is the *Marching Cubes* method introduced by Lorensen and Cline [1]. The marching cubes method concentrated on the approximation of the isosurface inside the cells rather than on efficient locations of the involved cells. To this end, the marching cube method scans the *entire* cell set, one cell at a time. The novelty of the method is the way in which it decides for each cell *whether* the isosurface intersects that cell and if so, *how* to approximate it.

A.2 Octrees

The marching cubes method did not attempt to optimize the time needed to search for the cells that actually intersect the isosurface. This issue was later addressed by Wilhelms and Gelder [2], who employed an octree, effectively creating a 3D hierarchical decomposition of the cell set, C. Each node in the tree was tagged with the minimum and maximum values of the cells it represents. These tags, and the hierarchical nature of the octree, enable one to trim off sections of the tree during the search and thus restrict the search to only a portion of the original geometric space. Wilhelms and Gelder did not analyze the time complexity of the search phase of their algorithm. However, octree decompositions are known to be sensitive to the underlying data. If the underlying data contains some fluctuations or noise, most of the octree will have to be traversed. Fig. 13 is an example for such a data set, which ultimately undermines any geometric decomposition scheme. In Appendix A we present an analysis of the octree algorithm and show that the algorithm has a worst case complexity of $O(k + k \log n/k)$. Finally, octrees have primarily been applied to structured grids and are not easily adapted to deal with unstructured grids.

A.3 Extrema Graphs

Recently, Itoh and Koyamada [3] presented a new method for generating isosurfaces over unstructured grids using *extrema graphs.*

The search starts at a *seed* cell known to intersect the isosurface, and propagates recursively to its neighbor cells. Knowing how the isosurface intersects the current cell enables the algorithm to move only to those neighbor cells that are guaranteed to intersect the isosurface.

In order to find such a seed cell, Itoh and Koyamada employed extrema graphs. The nodes of these graphs are those cells that include local extrema vertices. Each arc in the graphs has a list of the cells connecting its two end nodes.

Given an isovalue, the extrema graph is first scanned to located arcs that span across the isovalue. The cells in each such arc's list are then scanned sequentially until a seed cell is found. Boundary cells must also be traversed; hence the complexity of the algorithm is at best the size of the boundary list, which Itoh and Koyamada estimate as $O(n^{2/3})$.

Our analysis shows that the number of arcs can be of $O(n)$ in the worst case. Such a case occurs when the data exhibits small perturbations such that each node is a local extrema. In such a case, the numbers of arcs in the extrema graph can be equal to the number of cells, though each arc will contain only a single cell.

Storage requirements for the extrema graph method can be high, since the propagation search requires four links from each cell to its neighbors in addition to the maximum and minimum values of its vertices. In addition, the algorithm uses a queue during the propagating search, yet the maximum required size of the queue is unknown in advance.

B. Value Space Decomposition

Decomposing the value space, rather than the geometric space, has two advantages. First, the underlying geometric structure is of no importance, so this decomposition works well with unstructured grids. Second, for a scalar field in 3D, the dimensionality of the search is reduced from three to only two.

B.1 The Span Filter

A key issue in isosurface extraction is the size of the data set. Gallagher [5] addressed this issue by scanning the data set and generating a compressed representation suitable for isosurface extraction. The range of data values is divided into sub-ranges, termed *buckets.* Each cell is then classified based on the bucket its minimum value resides in and on how many buckets the cell's range spans, i.e. the *span* of the cell. Cells are then grouped according to their span, and within each such group the cells are further grouped according to their starting bucket. In each such internal group, the representation is compressed according to a unique id assigned to each cell. Rather than requiring a span list for every possible span length, the method uses one span list to catch all the cells that span more than a

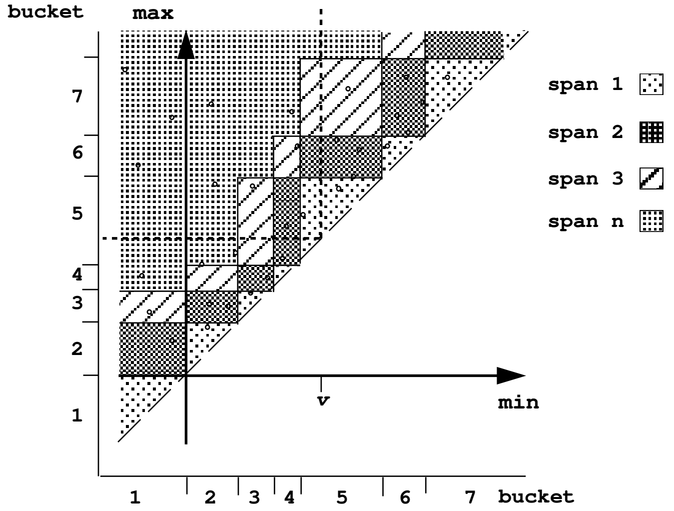

Fig. 2. Span Filter. Shown is the *ad hoc* division of a field's range into subranges called buckets. Each point, which represents a data cell, is then assigned a min and max bucket, based upon the point's min and max coordinate. The points are then grouped into spans based upon the difference between their assigned buckets' numbers. Span n represents all the spans with index larger than some predefined index, i.e. 3 in this example.

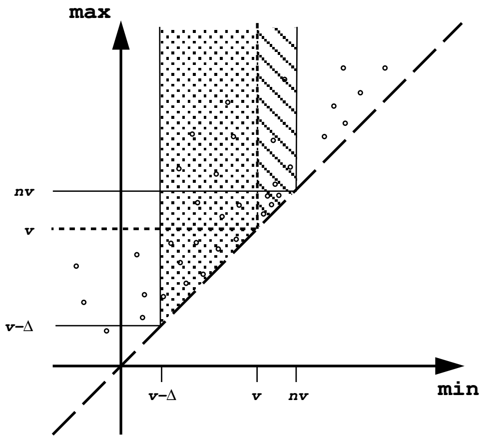

Fig. 3. Active List. The doted area represents the points that are initially put into the active list. The points in the doted area below the horizontal line, v are then removed from the active list. When the new isovalue, nv is close to the current isovalue, v, only the points in the striped area are added to the active list. The points below the horizontal nv line, within both the striped and the dotted areas, are then removed from the active list.

predefined number of buckets.

Fig. 2 depicts the span filter organization over the *span space*. Note that the compression over the cells' id is not shown. For a given isovalue, v, the cells that intersect the isosurface are those that lie above and to the left of the dashed line.

The use of this perspective stresses the importance of the first division into buckets. The entire organization of the domain is controlled by only one set of parameters, the position of the original buckets. While this may help to ensure even distribution in the first span, it does not provide control over the distribution of the cells in the other spans. Furthermore, this division is not automated and has to be crafted by trial and error for each new data set. Finally, the search algorithm has a complexity of $O(n)$ in time.

B.2 The Active List

A different approach was taken by Giles and Haimes [4]: to find the cells that intersect an isosurface incrementally. Once an isosurface is found, then a neighbor isosurface, with an isovalue close to the first one, can be found with minimal effort.

The algorithm is based on two cell lists ordered by the cell's minimum and maximum values and on Δ, the global maximum range of any of the cells. When an isovalue is first given, or if the change from the previous value is greater than Δ, then an active cell list is formed. The active list is first initialized with all the cells with a minimum value between the given isovalue, v, and $v - \Delta$, by consulting the minimum list. The active list is then purged of the cells with a maximum value less then v. If the isovalue is changed by less then Δ, then the active list is augmented with the cells that lie between the previous isovalue, v and the new one, nv. The new cells are found by using one of the two ordered lists, based upon whether the change was positive or negative. The active list is then purged again for the cells that do not intersect the isosurface.

Fig. 3. depicts Giles' and Haimes' algorithm over the span space. Though the algorithm does not explicitly partition the space in advance, the use of the global maximum cell span, Δ, does the same thing implicitly, as the width of the area that needs to be scanned is constant. When the change in the isovalue is greater than Δ, the algorithm must linearly scan *all* the cells in the range $(nv - \Delta, nv)$. Since Δ depends on the data set, the algorithm has no control over the size of the scanned list. In two of our test cases, *Heart* and *Brain*, there are few cells on the boundary that have a very large span. This causes Δ to be so large that the algorithm must linearly scan approximately half of the data set. On the other hand Δ might be too small such that the neighborhood search may not be used at all. Using the span perspective, Fig. 3, we can see that when the isovalue is changed from v to nv the algorithm will scan all the cells in the striped band but will then discard those cells that are in the lower triangle of that band. This triangle is usually the most dense part of the band, so that a large number of cells must be scanned and then discarded. If one scans across the entire range of the data set, a typical change in the isovalue will be larger than 0.5%, while, for a large data set, Δ will be much smaller, again not taking advantage of neighboring isosurfaces. Finally, the algorithm's complexity is still $O(n)$ in time.

B.3 Sweeping Simplices

Recently, two of the authors, Shen and Johnson [6], developed the *sweeping simplices* method for extracting isosurfaces from unstructured three-dimensional meshes. Their algorithm utilizes both coherence between adjacent isosurfaces and explicit space decomposition.

Sweeping simplices uses two ordered cell lists, a *sweep list* and a *min list*. Each element in the sweep list contains a pointer to a cell, the cell's maximum value, and a flag. The sweep list is then sorted according the cell's maximum value. The min list contains the minimum value for each cell as well as a pointer to the corresponding element in the sweep list and is ordered by the minimum values. The initialization step requires a time of $O(n \log n)$.

Given an isovalue, the sweeping simplices algorithm *marks* all the cells that have a minimum value less than the given isovalue using the min list by setting the corresponding flag in the sweep list. If an isovalue was previously given, then the min list is traversed between the previous isovalue and the new one. The corresponding flags in the sweep list are then set or reset based on whether the new isovalue is greater or smaller than the previous isovalue.

Once the flags are changed, the sweep list is traversed starting at the first cell with a maximum value greater than the new isovalue. The cells that intersect the isosurface are those cells for which their corresponding flag is set. The complexity of the algorithm is $O(n)$ in both time and space.

The sweeping simplices algorithm uses a hierarchical data decomposition. At the lowest level, the range of data values is subdivided into several subgroups. Other levels are created recursively by grouping consecutive pairs from the previous level. At the top level there exists a single subgroup with the range as the entire data set. The cells are then associated with the smallest subgroup that contains the cell. Each subgroup is then associated with a min and sweep list as described before. Isosurface extraction is accomplished by selecting for each level the subgroup that contains the given isovalue and performing the search using its min and sweep lists.

The space decomposition for the sweeping simplices algorithm, as well as the marked cells for an isovalue pv, is shown in Fig 4. The full dots are the marked cells. When a new isovalue is selected, all the cells that lie between the vertical lines pv and v are first marked. The cells that intersect the isosurface are those marked cells that lie above the horizontal line at v. Though sweeping simplices is faster than the active list algorithm and does not depend on a global Δ, its space decomposition is not optimal. Each of the groups whose range intersect the isovalue lines, Fig. 4, must be linearly scanned and each such group contains an area outside the target isosurface region. We remark that using the span space perspective, the second author recently devised a more efficient space decomposition algorithm that improved the overall performance of the sweeping simplices algorithm.

B.4 Summary of Existing Methods

Previous value space decomposition algorithms use a wide range of terminology and approaches. The use of the span space provides a common ground on which these methods can be compared. In effect, it was shown that these methods use very similar approaches both in searching and in space decomposition. All of these methods have complexity of $O(n)$ in both time and memory requirements.

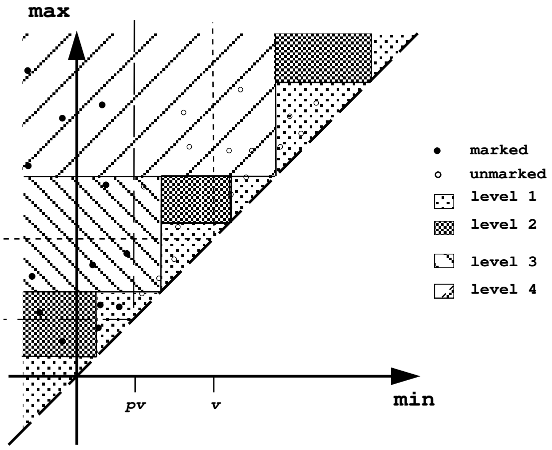

Fig. 4. Sweeping Simplices. The range of the field is divided into subranges that are, in turn, organized into levels. See text for further details.

IV. The New Algorithm

A common obstacle for all the interval methods was that the intervals were ordered according to *either* their maximum or their minimum value. Both the sweep algorithm and the min-max attempted to tackle this issue by maintaining two lists of the intervals, ordered by the maximum and minimum values. What was missing, however, was a way to combine these two lists into a single list.

In the following, we present a solution to this obstacle. Using the span space as our underlying domain, we employ a kd-tree as a means for simultaneously ordering the cells according to their maximum and minimum values.

A. *Kd-Trees*

Kd-trees were designed by Bentley in 1975 [7] as a data structure for efficient associative searching. In essence, kd-trees are a multi-dimensional version of binary search trees. Each node in the tree holds one of the data values and has two sub-trees as children. The sub-trees are constructed so that all the nodes in one sub-tree, the *left* one for example, hold values that are less than the parent node's value, while the values in the *right* sub-tree are greater than the parent node's value.

Binary trees partition data according to only one dimension. Kd-trees, on the other hand, utilize multidimensional data and partition the data by alternating between each of the dimensions of the data at each level of the tree.

B. *Search over the Span Space Using Kd-Tree*

Given a data set, a kd-tree that contains pointers to the data cells is constructed. Using this kd-tree as an index to the data set, the algorithm can now rapidly answer isosurface queries. Fig. 5 depicts a typical decomposition of a span space by a kd-tree.

Construction

The construction of the kd-trees can be done recursively in optimal time $O(n \log n)$. The approach is to find the

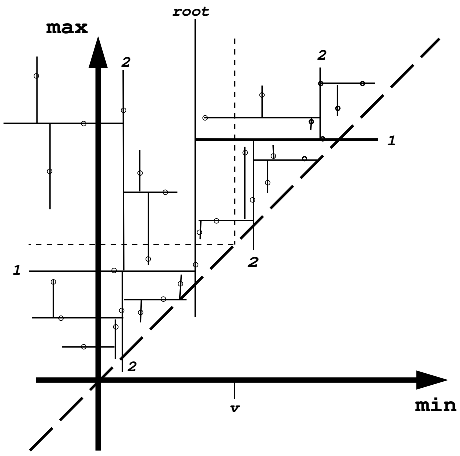

Fig. 5. Kd Tree. The lines represent the structure of the kd-tree. The vertical line *root* represents the first split of the span space along the min coordinate. The next split, at level 1, is represented by two horizontal lines that split the two major subregions along the max coordinate. At level 2 of the tree, the split of the, now four subspaces, is again along the min coordinate. The processes continues until all of the points are accounted for.

median of the data values along one dimension and store it at the root node. The data is then partitioned according to the median and recursively stored in the two sub-trees. The partition at each level alternates between the min and max coordinates.

An efficient way to achieve $O(n \log n)$ time is to recursively find the median in $O(n)$, using the method described by Blum *et al.*[8], and partition the data within the same time bound.

A simpler approach is to sort the data into two lists according to the maximum and minimum coordinates, respectively, in order $O(n \log n)$. The first partition accesses the median of the first list, the *min* coordinate, in constant time, and marks all the data points with values less than the median. We then use these marks to construct the two sub groups, in $O(n)$, and continue recursively.

Though the above methods have complexity of $O(n \log n)$, they do have weaknesses. Finding the median in optimal time of $O(n)$ is theoretically possible yet difficult to program. The second algorithm requires sorting two lists and maintaining a total of four lists of pointers. Although it is still linear with respect to its memory requirement, it nevertheless poses a problem for very large data sets.

A simple (and we think elegant) solution is to use a Quicksort-based selection [9]. While this method has a *worst case* of $O(n^2)$, the *average* case is only $O(n)$. Furthermore, this selection algorithm requires no additional memory and operates directly on the tree. We note that this algorithm performed *at least* four time faster on all of our application data sets in section VII than the two sorted lists algorithm.

Pseudo code for the kd-tree construction is given in Fig. 6.

It is clear that the kd-tree has one node per cell, or span point, and thus the memory requirement of the kd-tree is

```
build-kd-tree( array, size )
{
  // recursive build
  build (array, size, min );
}

build ( array, size, criterion )
{
  // criterion is either min or max coordinate

  if ( size < 2 ) return;
  partition( array, size, criterion );
  build( array, size/2, other-criterion );
  build( array+1+size/2, (size-1)/2,
         other-criterion );
}


partition( array, size, criterion )
{
  Use Quicksort partition algorithm to
  rearrange the array, based on the given
  criterion, such that the median element
  is in array[size/2] and all the elements
  less then the median are in array[0..size/2-1].
}
```

Fig. 6. Kd-Tree Construction.

$O(n)$.

Query

Given an iso-value, v, we seek to locate all the points in Fig. 1 that are to the *left* of the vertical line at v and are *above* the horizontal line at v. We note that we do not need to locate points that are *on* these horizontal or vertical lines if we assume non-degenerate cells, for which minimum or maximum values are not unique. We will remove this restriction later.

The kd-tree is traversed recursively by comparing the iso-value to the value stored at the current root alternating between the root's minimum and maximum values at odd and even levels. If the root node is to the right (below) of the iso-value line, then only the left (right) sub-tree should be traversed. Otherwise, *both* sub-trees should be traversed recursively. Furthermore, in this last case the root's other value should also be compared to the given iso-value to determine if the corresponding cell should be triangulated. For efficiency we define two search routines, *search-min-max* and *search-max-min*. The dimension we currently checking is the first named, and the dimension we still need to search is named second. The importance of naming the second dimension will be evident in the next section, when we consider optimizing the algorithm.

Following is a short pseudo-code for the min-max routine.

```
search-min-max( iso-value, root )
{
```

```
  if ( root.min < iso-value ) {
     if ( root.max > iso-value )
        construct polygon(s) from root's cell
     search-max-min( iso-value, root.right );
  }

  search-max-min( iso-value, root.left );
}
```

Estimating the complexity of the query is not straightforward. Indeed, the analysis of the worst case was developed by Lee and Wong [10] only several years after Bentley introduced kd-trees. Clearly, the query time is proportional to the number of nodes visited. Lee and Wong analyzed the worst case by constructing a situation where all the visited nodes are not part of the final result. Their analysis showed that the worst case time complexity is $O(\sqrt{n}+k)$. The average case analysis of a region query is still an open problem, though observations suggest it is much faster than $O(\sqrt{n}+k)$ [9], [11]. In almost all typical applications $k \sim n^{2/3} > \sqrt{n}$, which suggests a complexity of only $O(k)$. On the other hand, the complexity of the isosurface extraction problem is $\Omega(k)$, because it is bound from below by the size of the output. Hence, the proposed algorithm, NOISE, is optimal, $\theta(k)$, for almost all cases and is near optimal in the general case.

Degenerate Cells

A degenerate cell is defined as a cell having more then one vertex with a minimum or maximum value. When a given iso-value is equal to the extrema value of a cell, the isosurface will not intersect the cell. Rather, the isosurface will touch the cell at a vertex, an edge, or a face, based on how many vertices share that extrema value. In the first two cases, vertex or edge, the cell can be ignored. The last case is more problematic, as ignoring this case will lead to a hole in the isosurface. Furthermore, if the face is not ignored, it will be drawn twice.

One solution is to perturb the isovalue by a small amount, so that the isosurface will intersect the inside of only one of those cells. Another solution is to check *both* sides of the kd-tree when such a case occurs. While the direct cost of such an approach is not too high as this can happen at most twice, there is a higher cost in performing an equality test at each level. We note that in all the data sets we tested there was not a single case of such a degeneracy.

V. Optimization

The algorithm presented in the previous section is not optimal with regards to both the memory requirement and search time. We now present several strategies to optimize the algorithm.

A. Pointerless Kd-Tree

A kd-tree node, as presented previously, must maintain links to its two sub-trees. These links introduce a high cost in terms of memory requirements. To overcome this deficiency, we note that in our case the kd-tree is completely balanced. At each level, one data point is stored at the node and the rest are equally divided between the two sub-trees. We can, therefore, represent a pointerless kd-tree as a one-dimensional array of the nodes. The root node is placed at the middle of the array, while the first $n/2$ nodes represent the left sub-tree and the last $(n-1)/2$ nodes the right sub-tree, as shown in Fig. 7.

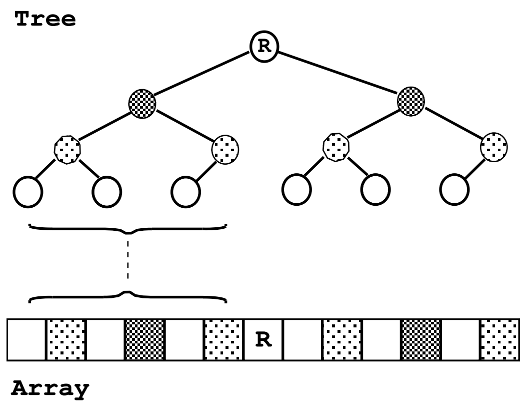

Fig. 7. Two representations of a kd-tree and the relative position of their nodes.

The memory requirement, per node, for a pointerless kd-tree reduces to two real numbers, for minimum and maximum values, and one pointer back to the original cell for later usage. Considering that each cell, for a 3D application with tetrahedral cells has pointers to four vertices, the kd-tree memory overhead is even less than the size of the set of cells.

The use of a pointerless kd-tree enables one to compute the tree as an off line preprocess and load the tree using a single read in time complexity of only $O(n)$. Data acquisition via CT/MRI scans or scientific simulations is generally very time consuming. The ability to build the kd-tree as a separate preprocess allows one to shift the cost of computing the tree to the data acquisition stage. Hence, reducing the impact of the initialization stage on the extraction of isosurfaces for large data sets.

B. Optimized Search

The search algorithm can be further enhanced. Let us consider, again, the min-max (max-min) routine. In the original algorithm, if the iso-value is less then the minimum value of the node, then we know we can trim the right sub-tree. Consider the case where the iso-value is greater then the node's minimum coordinate. In this case, we need to traverse *both* sub-trees. We have no new information with respect to the search in the right sub-tree, but, for the search in the left sub-tree we *know* that the minimum condition is satisfied. We can take advantage of this fact by skipping over the odd levels from that point on. To achieve this, we define two new routines, *search-min* and *search-max*. Adhering to our previous notation, the name search-min states that we are only looking for a minimum value.

```
search_min_max( iso_value, root ) {
  if ( root.min < iso_value ) {
     if ( root.max > iso_value )
        construct polygon(s) from root's cell;
     search_max_min( iso_value, root.right );
     search_max( iso_value, root.left );
  } else
     search_max_min( iso_value, root.left );
}
search_min( iso_value, root ) {
  if ( root.min < iso_value ) {
     construct polygon(s) from root's cell;
     search_skip_min( iso_value, root.right );
     collect( root.left );
  } else
     search_skip_min( iso_value, root.left );
}
search_skip_min( iso_value, skip_node ) {
  if ( skip_node.min < iso_value )
     construct polygon(s) from skip_node's cell;
  search_min( iso_value, skip_node.right );
  search_min( iso_value, skip_node.left );
}
collect( sub_tree ) {
  for (each leaf node)
    construct polygon(s) for leaf's cell.
  Note: the leaf nodes are organized
         sequentially and thus there is
        no need to descend this subtree.
}
```

Fig. 8. Optimized Search

Examining the search-min routine, we note that the maximum requirement is already satisfied. We do not gain new information if the isovalue is less than the current node's minimum and again only trim off the right sub-tree. If the iso-value is greater than the node's minimum, we recursively traverse the right sub-tree, but with regard to the left sub-tree, we now know that all of its points are in the query's domain. We therefore need only to *collect* them. Using the notion of pointerless kd-tree as proposed in Sec V-A, any sub-tree is represented as a *contiguous* block of the tree's nodes. Collecting all the nodes of a sub-tree requires only sequentially traversing this contiguous block.

Pseudo code of the optimized search for the odd levels of the tree, i.e. searching for minima is presented in Fig. 8. The code for even levels, searching for maxima, is essentially the same and uses the same collect routine.

C. Count Mode

Extracting isosurfaces is an important goal, yet in a particular application one may wish only to know how many cells intersect a particular isosurface. Knowing the number of cells that intersect the isosurface can help one give a rough estimate of the surface area of the isosurface on a structured grid and on a "well behaved" unstructured grid. The volume encompassed by the isosurface can also be estimated if one knows the number of cells that lie inside the isosurface as well as the number of cells that intersect it.

The above algorithm can accommodate the need for such particular knowledge in a simple way. The number of cells intersecting the isosurface can be found by incrementing a counter rather than constructing polygons from a node and by replacing collection with a single increment of the counter with the size of the sub-tree, which is known without the need to traverse the tree. To count the number of cells that lie inside the isosurface, one need only look for the cells that have a maximum value below the iso-value.

The worst case complexity of the count mode is only $O(\sqrt{n})$. A complete analysis is presented in Appendix B. It is important to note that the count mode does not depend on the size of the isosurface. We shall show in Section VII that such a count is extremely fast and introduces no meaningful cost in time. The count mode thus enables an application to quickly count the cells that intersect the isosurface and allocate and prepare the appropriate resources *before* a full search begins.

D. Neighborhood Search

The Sweeping Simplices and the Active List algorithms were designed to take advantage of coherence between isosurfaces with close isovalues. We now present a variant of the proposed algorithm that also takes advantage of such coherence.

By examining Fig. 10 we see that if an isovalue pv is changed to v, then the set of cells that intersect the new isosurface can be generated by adjusting the current set of cells. In essence, if $v > pv$ then we need to remove the cells that lie in the bottom rectangle and add those that lie in the right rectangle. If $v < pv$ the add and remove roles of these rectangles are flipped. As opposed to the previous methods, which decompose the space specifically for small changes in the isovalue, we can use the kd-tree decomposition as is. This, in turn, means that at any time, either the regular or the neighborhood search can be performed over the same data structure and thus we can choose which one will likely be the best one based on the current estimation. The new set of cells is achieved by performing two searches. First the kd-tree is searched for cells that need to be removed. A second search is then performed to find new cells to add to the list. Fig. 9 depicts a pseudo code for a part of the second search.

The neighborhood search can benefit when the change in the isovalue is small and only a small number of cells needs to be added or removed, especially in the count mode. However, there are several disadvantages in using this type of search, as was the case in previous methods. First, an active cell list must be maintained that adds more overhead both in time and memory. Second, each node in the kd-tree must maintain yet another pointer to the cell entry in the active list so that it can be removed quickly without traversing the active list. Finally, if the number of cells that belong to *both* the current and the new cell list is small, the effort to find the new isosurface is doubled.

```
near-search-min-max( pv, v, node )
{
  if ( node.min < pv )
     near-search-max-min( pv, v, node.right );
  else
  if ( node.min > v )
     near-search-max-min( pv, v, node.left );
  else
  {
     if ( node.max > v )
        add node;
     near-search-max-min( pv, v, node.right );
     near-search-max-min( pv, v, node.left );
  }
}
```

Fig. 9. Neighborhood Search - Pseudo Code

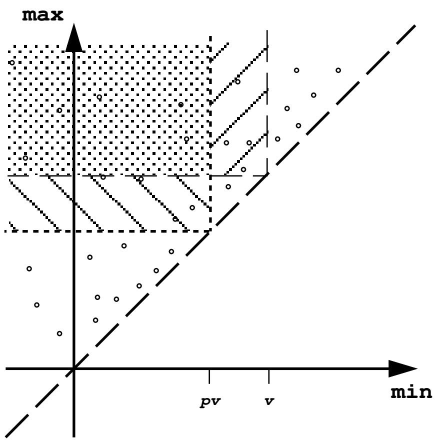

Fig. 10. Neighborhood Search. The points in the dotted area represent cells that are intersected by both the current isosurface and the new isosurface. The points (cells) in the right striped area should be added to the isosurface while the points (cells) in the lower striped area should be removed from the isosurface.

We remark that with the current performance of the algorithm and current available hardware, the bottle neck is no longer in finding the isosurface or even computing it, but rather in the actual time it takes to display it.

VI. Triangulation

Once a cell is identified as intersecting the isosurface, we need to approximate the isosurface inside that cell. Toward this goal, the marching cubes algorithm checks each of the cell's vertices and marks them as either *above* or *below* the isosurface. Using this information and a lookup table, the algorithm identifies the particular way the isosurface intersects the cell. The marching cubes, and its many variants, are designed for structured grids though they can be applied to unstructured grids as well.

We propose a new algorithm for unstructured grids of tetrahedral cells. We first note that if an isosurface intersects *inside* a cell, then the the vertex with the maximum value *must* be above the isosurface and the vertex with the minimum value *must* be below it.

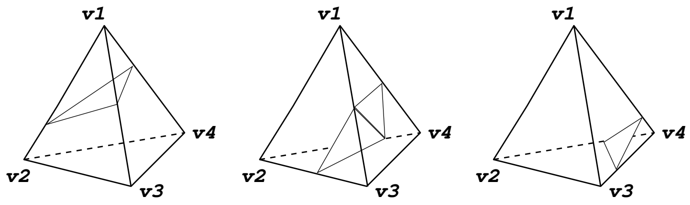

Fig. 11. Triangulation. The vertices are numbered according to ascending values.

To take advantage of this fact, we reorder the vertices of a cell according to their ascending values, say v1 to v4, a priori, in the initialization stage. When the cell is determined to intersect the isosurface, we need only to compare the iso-value against *at most* the two middle vertices. Since there are only three possible cases: only v1 is *below* the isosurface, only v4 is *above* the isosurface, or {v1,v2} are below and {v3, v4} are above see Fig. 11. Moreover, the order of the vertices of the approximating triangle(s), such that the triangle(s) will be oriented correctly with respect to the isosurface, is known in advance at no cost. We can further take advantage of the fact that there are only four possible triangles for each cell and compute their normals a priory. This option can improve the triangulation time dramatically yet it comes with a high memory price tag.

VII. Results

To evaluate the proposed algorithms, we have done extensive tests on various data sets. The tests were carried on SGI (R4400, 150MHz) workstations with 256Mb and 640Mb of memory.

A. The data sets

We have used several data sets from a variety of sources. Table I shows the characteristics of these models. The first three data sets consists of bio-electric field problems solved using the finite element method on unstructured tetrahedral grids, Fig. 14, 15, 16. *Head* is a 128^3 MRI scan of a human head, Fig. 12. The FD, Fluid Dynamics, data set is computed from a 256^3 spectral CFD simulation, Fig. 13. We also used sub-sampled sets of this large data set of sizes 32^3, 64^3 and 128^3 [1].

TABLE I
Data Sets

	Source	Type	Vertices	Cells
Heart	FEM	U-grid	11504	69892
Torso	FEM	U-grid	201142	1290072
Brain	FEM	U-grid	74217	471770
Head	MRI	S-grid	2M	2048383
FD-32	FEM	S-grid	32K	29791
FD-64	FEM	S-grid	256K	250047
FD-128	FEM	S-grid	2M	2048383
FD-256	FEM	S-grid	16M	16581375

[1] NOTE: We will submit color figures for the final paper.

B. Benchmarks

The algorithm was tested both with respect to CPU run time and its complexity relative to a given data set. Each test included 1000 random value isosurface extractions. Table II shows the distribution of the number of cells in the isosurfaces for the different models. The *Brain* model is an example of a non-uniform cell size and position distribution. Some of the cells had very large span that would have caused worst-case performance in previous isosurface extraction algorithms. We performed two tests on this model first using iso-values from the entire model domain and a second checking only a small dense area.

In this paper, we concentrated on finding the cells that intersect an isosurface and performing fast triangulation on tetrahedral cells. We therefore did not measure the triangulation of the structured grid model. For these data sets we issued a call to an empty stub function for each cell that intersects the iso-surface, therefore introducing some cost per intersected cell.

TABLE II
ISOSURFACE STATISTICS

	Cells in Isosurfaces		
	min	max	avg
Heart	0	23087	1617
Torso	32	31011	8001
Brain:			
partial	5287	26710	10713
full	12	14756	25
Head	8	610291	61091
FD-32	0	28541	3074
FD-64	0	230562	24720
FD-128	0	1680341	172247
FD-256	8	11172708	1088128

TABLE III
PERFORMANCE STATISTICS

	Isosurface size	Nodes checked	Overhead	Max. collected
Heart	1617	687	473	17472
Torso	8001	3487	2679	20156
Brain :				
partial	10713	2295	1570	14742
full	25	2043	2020	7370
Head	61091	4568	3735	512095
FD-32	3074	550	410	7447
FD-64	24720	1547	834	62511
FD-128	172247	4489	3405	512095
FD-256	1088128	12787	6940	4145343

C. Analysis

Table III shows the performance of the algorithm with respect to the size of an average isosurface. The first column was taken verbatim from Table II. The *Nodes Checked* column represents the average number of tree nodes that were actually examined by the algorithm of which *Overhead* were not part of the final isosurface. For example, the average isosurface in the *FD-128* case intersected $172,247$ cells, yet the algorithm had to examine *only* $4,489$ tree nodes in order to locate these cells. Out of the $4,489$ nodes that were checked $3,405$ nodes did not intersect the isosurface and there for represent an *overhead* in some sense. A key point in the algorithm is its ability to locate large groups of intersected cells i.e. large subtrees in which all of their nodes represent cells that are intersected by the isosurface. Once such a subtree is located, there is no need to traverse this subtree as its leaf nodes form a continuous block. The largest such subtree that was found in a a paticular data set is depicted under the *Collected* column of the table. In the case of our previous example, *Fd-128*, the largest such subtree contained $512,095$ nodes.

TABLE IV
CPU TIME

	Build	Count		Search[a]	
			per cell		per cell
	sec	*msec*	*nano-sec*	*msec*	*nano-sec*
Heart	7.6	0.4	5.7	7.0	43
Torso	27.1	2.2	1.7	43.8	55
Brain :	7.6				
partial		1.5	3.2	53.9	50
full		1.0	2.1	1.1	440
Head	35.2	3.0	1.4	31.4	15
FD-32	0.2	0.3	10.4	1.0	34
FD-64	2.3	0.9	3.5	7.7	31
FD-128	22.6	2.9	1.4	69.2	34
FD-256	203.8	8.9	0.5	420.4	25

[a] Search times include triangulation for unstructured grids only.

The algorithm consistently examined many fewer nodes than the size of the extracted isosurface. The only exception was the full Brain data set where the average isosurface was more or less empty. Even in this pathological case, the number of cells that were examined was small, only 0.43%. This is a case where the algorithm is not optimal as $k < \sqrt{n}$, yet the overhead is negligible. Overall, the overhead of examining extra nodes was kept at a minimum and the collection scheme achieved excellent results.

The complexity of the *search phase* was kept at $3\sqrt{n}$, which does not depend on the size of the resulting isosurface as predicted by the count mode analysis. CPU run time is shown in Tab. IV. The initialization step is measured in seconds while the *count* and *search* are in milliseconds. All numbers represent the average run time per query. The search includes triangulation for the unstructured grid data sets only, using the proposed fast triangulation algorithm. The time requirements for the count mode was kept to a few milliseconds, even for very large data sets with correspondingly large numbers of isosurfaces. The search optimization has clearly benefited from the collect routine, as is evident by the large collected blocks.

The performance of the algorithm should be viewed with respect to its main goal, that is, locating the cells that intersect the isosurface. In this respect, i.e. the *count* mode, the CPU time requirements were as low as a few milliseconds - even for large data sets and exhibit complexity of only $O(\sqrt{n})$, i.e. no dependency on the size of the isosurface was noticed. The search mode CPU time is clearly dominated by the size of the isosurface, as each intersected cell must be examined and triangulated. In the case of the unstructured grid datasets, the entire process of search and triangulation was about 50ms. However, for the large structured grid datasets, the average size of the isosurfaces was much larger and caused the total time to increase to approximately 0.8 seconds.

VIII. Conclusions

We presented the "Near Optimal IsoSurface Extraction" (NOISE) algorithm, which has a worst-case performance of $O(\sqrt{n}+k)$. The algorithm is *near optimal* in the sense that for the typical case, $k > \sqrt{n}$, NOISE is optimal, while for the rest of the cases the overhead is negligible. The memory requirement for NOISE is $O(n)$, while the preprocess step has a complexity of $O(n \log n)$ and can be performed offline. If the preprocessing is done offline, its results can be loaded in $O(n)$.

The algorithm performs well for large and small data sets and for any size of isosurface. The number of cells that intersect an isosurface can also be found in $O(\sqrt{n})$ time, which enables fast rough estimates of the surface area and the corresponding volume encompassed by the isosurface.

We were able to create the NOISE algorithm by projecting the data onto a new space, termed the *span space*, which, in turn, lends itself to a simple decomposition utilizing kd-tree. Furthermore, the span space can serve as a common ground on which other methods can be compared and analyzed.

We also presented a fast triangulation scheme based on a one time pre-process reorganization of the cells' vertices.

Acknowledgments

This work was supported in part by the National Science Foundation and the National Institutes of Health. The authors would like to thanks K. Coles and J. Painter for their helpful comments and suggestions. We wish to thank the Los Alamos National Laboratory for the the use of their facilities and the *Head* data set. The *FD* data set is courtesy of Shi-Yi Chen of LANL. Furthermore, we appreciate access to facilities that are part of the NSF STC for Computer Graphics and Scientific Visualization.

Appendices

A. Worst Case Analysis for Octree Isosurface Extraction

Wilhelms and Gelder did not analyze the time complexity of their octree-based isosurface extraction algorithm, Section III-A.2. We now present a worst-case analysis of their method.

We first note that the octree used by Wilhelms and Gelder is derived from the geometry of the data set and is only *augmented* by the minimum and maximum values of the cells in the tree. As such, the octree relies solely on geometry to group cells with close field values. On the other hand, the octree is guaranteed to be balanced. Also note that the data cells occur only on the leaves of the tree.

For simplicity, consider first the 1D case of a binary tree with n leaves. For a given k, we seek one of the groups of k leaves with the highest cost to locate. For $k = 1$, the cost is $\log n$; this suggests an estimate of $O(k \log n)$ for the worst case. This is clearly an overestimate as many segments of the paths to these k cells are shared. When $k = 2$, the two paths from the root must share several intermediate nodes. The maximum cost will occur when only the root node is shared. Therefore,

$$\begin{aligned} T(n,1) &= \log(n) \\ T(n,k) &= 1 + 2T(n/2, k/2), \end{aligned}$$

which, for $k = 2^m$, leads to

$$T(n,k) = k - 1 + k \log\left(\frac{n}{k}\right).$$

As an example, $T(n,n) = 2n - 1$, since the a binary tree with n leaves has $n - 1$ internal nodes.

The general case for a d-dimensional tree follows immediately from the binary case. Let $p = 2^d$,

$$\begin{aligned} T_d(n,1) &= \log_p n \\ T_d(n,k) &= 1 + pT(n/p, k/p) \end{aligned}$$

Let $q = \log_p k$. The solution to the recursive formula is

$$\begin{aligned} T_d(n,k) &= \frac{p^q - 1}{p - 1} + p^q \log_p\left(\frac{n}{k}\right) \\ &= \frac{k-1}{p-1} + \frac{k}{d}\log\left(\frac{n}{k}\right). \end{aligned}$$

For the special case of octree, $d = 3$, we have

$$T_3(n,k) = \frac{k-1}{7} + \frac{k}{3}\log\left(\frac{n}{k}\right)$$

and a complexity of $O(k + k\log(\frac{n}{k}))$.

B. Performance Analysis for the Count Mode

A node in a kd-tree holds information regarding only the value used to split the current tree. This forces a search algorithm always to traverse at least one subtree. The best case performance for the count mode is thus $O(\log n)$.

We now examine the worst case complexity of the count mode. Referring to the optimized version, section V-A, we find two cases. When the isovalue is less than the value at the root of the tree we need to traversed only one subtree. Otherwise, both subtrees are traverse, yet for one of them we now know that the min or max condition is satisfied. Clearly the worst case involves the second case,

$$\begin{aligned} T(1) &= 1 \qquad (2)\\ T(n) &= 1 + T(n/2) + T_m(n/2). \qquad (3) \end{aligned}$$

For the case where the min or max condition is satisfied there are again two cases. These cases, however, are different from each other only with respect to whether one of the subtree is completely empty or full. In both these cases, only one subtree is descended. Moreover, the next level of this subtree can be skipped and the algorithm descends directly to both sub-subtrees. Note that the root of the subtree still need to be checked. Therefore,

$$\begin{aligned} T_m(1) &= 1\\ T_m(n) &= 1 + 2T_m(n/4)\\ &= \sum_{i=0}^{\log_4 n} 2^i\\ &= 2^{\log_4(n)+1} - 1\\ &\leq 2\sqrt{n}. \qquad (4) \end{aligned}$$

Substituting Eq. 4 in Eq. 3 and using Eq. 2 we get,

$$\begin{aligned} T(n) &\leq 1 + 2\sqrt{n} + T(n/2)\\ &= \log n + 2\sqrt{n} \sum_{i=0}^{\log(n)-1} 2^{-i/2}\\ &= \log n + 2\sqrt{n}\frac{1-\sqrt{2/n}}{1-1/\sqrt{2}}\\ &\leq \log n + 6\sqrt{n}. \end{aligned}$$

Hence a complexity of $O(\sqrt{n})$.

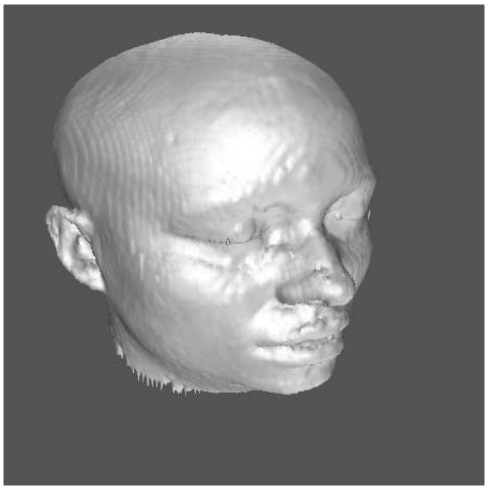

Fig. 12. Head: Iso-surface from a 128^3 MRI scan.

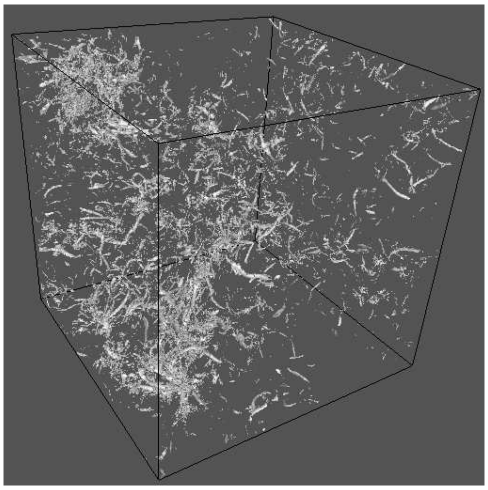

Fig. 13. Turbulent flow in a fluid dynamic simulation representing the magnitude of fluid velocity and showing the onset of turbulence. The elongated structures are vortex tubes.

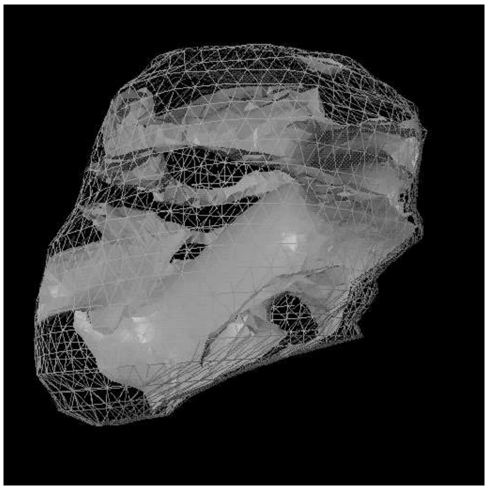

Fig. 14. Heart: Isosurfaces of constant voltage from a finite element simulation of cardiac defibrillation within the ventricles of the human heart.

References

[1] W.E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm", *Computer Graphics*, vol. 21, no. 4, pp. 163–169, July 1987.

[2] J. Wilhelms and A. Van Gelder, "Octrees for faster isosurface generation", *ACM Transactions on Graphics*, vol. 11, no. 3, pp. 201–227, July 1992.

[3] T. Itoh and K. Koyyamada, "Isosurface generation by using extrema graphs", in *Proceedings of Visualization '94*. 1994, pp. 77–83, IEEE Computer Society Press, Los Alamitos, CA.

[4] M. Giles and R. Haimes, "Advanced interactive visualization for CFD", *Computing Systems in Engineering*, vol. 1, no. 1, pp. 51–62, 1990.

[5] R. S. Gallagher, "Span filter: An optimization scheme for volume visualization of large finite element models", in *Proceedings of Visualization '91*. 1991, pp. 68–75, IEEE Computer Society Press, Los Alamitos, CA.

[6] H. Shen and C. R. Johnson, "Sweeping simplicies: A fast isosurface extraction algorithm for unstructured grids", *Proceedings of Visualization '95*, 1995, (to appear).

[7] J. L. Bentley, "Multidimentional binary search trees used for associative search", *Communications of the ACM*, vol. 18, no. 9, pp. 509–516, 1975.

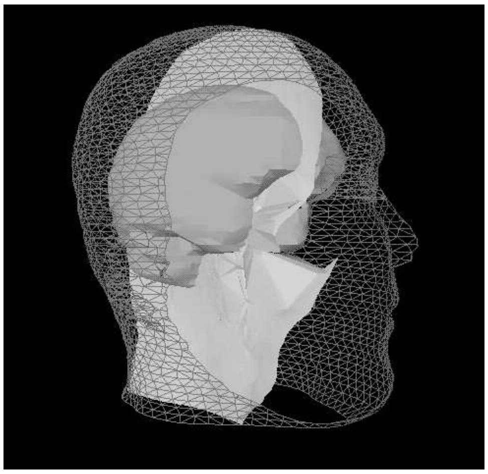

Fig. 15. Brain: An isosurface of constant voltage from a finite element simulation of temporal lobe epilepsy in a model of the human skull and brain.

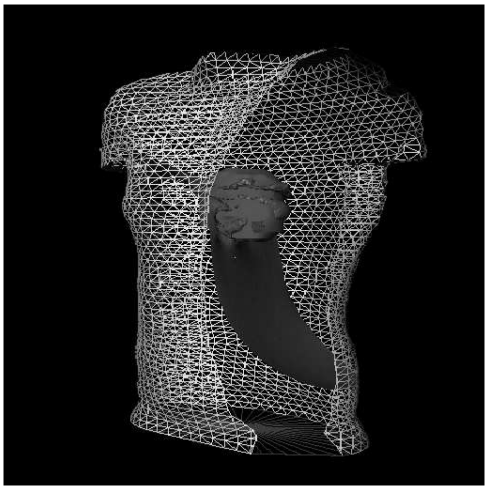

Fig. 16. Torso: An isosurface of constant voltage from a finite element simulation of the voltage distribution due to the electrical activity of the heart within a multi-chambered model of the human thorax.

[8] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection", *J. of Computer and System Science*, vol. 7, pp. 448–461, 1973.

[9] Sedgewick R., *Algorithms in C++*, Addison–Wesley, Massachusetts, 1992.

[10] D. T. Lee and C. K. Wong, "Worst-case analysis for region and partial region searches in multidimentional binary search trees and balanced quad trees", *Acta Informatica*, vol. 9, no. 23, pp. 23–29, 1977.

[11] J.L. Bentley and Stanat D. F., "Analysis of range searches in quad trees", *Info. Proc. Lett.*, vol. 3, no. 6, pp. 170–173, 1975.

Yarden Livnat Received a B.Sc. in computer science in 1982 from Ben Gurion University in the Negev, Israel and an M.Sc. cum laude in computer science from the Hebrew University, Israel in 1991. He is currently a Ph.D. candidate at the University of Utah working with the Scientific Computing and Imaging Research Group. His research interests include computational geometry, geometric modeling, scientific visualization and computer generated holograms.

Han Wei Shen is currently a Ph.D. candidate at the University of Utah working with the Scientific Computing and Imaging Research Group. He received his B.S. in 1988 from the National Taiwan University in Taipei, Taiwan, and his M.S. in 1992 from State University of New York at Stony Brook. His research interests include scientific visualization, computer graphics, and parallel rendering.

Christopher Johnson received his Ph.D. from the University of Utah in 1989. He is currently a member of the faculty in the Department of Computer Science at the University of Utah. His research interests are in the area of scientific computing. Particular interests include inverse and imaging problems, adaptive methods for partial differential equations, numerical analysis, large scale computational problems in medicine, and scientific visualization. In 1992, Professor Johnson was awarded a FIRST Award from the NIH, a National Young Investigator (NYI) Award from the NSF in 1994, and the Presidential Faculty Fellow (PFF) Award in 1995. He heads the Scientific Computing and Imaging (SCI) research group at the University of Utah.