Sweeping Simplices: A fast iso-surface extraction algorithm for unstructured grids

Han-Wei Shen and Christopher R. Johnson
Department of Computer Science
University of Utah
Salt Lake City, UT 84112.
E-mail: hwshen@cs.utah.edu and crj@cs.utah.edu

Abstract

We present an algorithm that accelerates the extraction of iso-surfaces from unstructured grids by avoiding the traversal of the entire set of cells in the volume. The algorithm consists of a sweep algorithm and a data decomposition scheme. The sweep algorithm incrementally locates intersected elements, and the data decomposition scheme restricts the algorithm's worst-case performance. For data sets consisting of hundreds of thousands of elements, our algorithm can reduce the cell traversal time more than 90% over the naive iso-surface extraction algorithm, thus facilitating interactive probing of scalar fields for large-scale problems on unstructured three-dimensional grids.

1 Introduction

Displaying iso-surfaces is an effective way to visualize three-dimensional scalar fields. Used to represent regions where the physical field is constant, iso-surfaces allow visualization with a full range of visual depth. By visualizing the spatial distributions of several iso-surfaces, scientists are able to obtain a greater understanding of a field's underlying structure; however, displaying multiple contours in a single frame has proved difficult. Dynamically probing the scalar field interactively by varying the iso-value has proven effective, but this method has been difficult to perform in real time. We present a new algorithm that overcomes the previous obstacles for interactive iso-surface extraction on unstructured grids.

The marching cubes algorithm proposed by Lorensen and Cline [1] has become a standard technique to generate iso-surfaces. Although the algorithm originates from medical applications, which typically use structured grids, the algorithm can be extended to problems using unstructured grids of tetrahedral elements. A naive implementation of the algorithm works as follows. Each cell in the field is traversed. Depending on the scalar values at the cell's corners, the geometry of surfaces inside each tetrahedron is determined. For a field represented by tetrahedral elements, there are only three different types of surface-cell intersection geometry, as shown in Figure 1. Although the naive algorithm is straightforward to implement, it is computationally inefficient. Given a particular iso-value, only a portion of cells in the field will intersect with the given value; for this reason, the traversal of the entire set of cells in the volume is unnecessary. Theoretically, the upper bound for the number of intersected cells is approximately $O(N^{2/3})$.

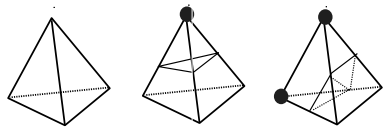

Figure 1: Surface-cell intersection geometry.

In this paper we present an algorithm that addresses the cell traversal problem to achieve fast iso-surface extraction within unstructured grids. The algorithm comprises two parts: a data decomposition scheme and a sweep algorithm. The data decomposition scheme minimizes the number of cells that need to be inspected for a particular iso-value. The sweep algorithm incrementally locates the intersected cells for the specified value. Our algorithm significantly reduces cell traversal time not only when the iso-value changes smoothly, but also when the iso-value changes dramatically.

In the following sections, we begin by surveying related work. We then describe our sweep algorithm and data decomposition scheme. Finally, we conclude with a detailed algorithm analysis and results of simulation

studies.

2 Related Work

Octrees: Wilhelms and Van Gelder [2] use an octree data structure to accelerate iso-surface extraction. The algorithm starts with a setup phase that creates the octree. Each node of the octree contains the maximum and minimum scalar values among the data elements in the sub-volume. When the user specifies an iso-value, the algorithm starts the surface-finding phase, which examines the volume by traversing from the root of the octree. All the sub-volumes with minimum values higher than the iso-value or maximum values lower than the iso-value are then excluded. When a leaf node is visited, this node's eight cells are examined to generate surfaces.

Octrees can substantially reduce cell traversal time because visits to non-intersected regions can be entirely avoided. However, octree data structures have primarily been implemented for data on structured grids. Moreover, octree methods are often sensitive to high-frequency noise within the data.

Active List: Giles and Haimes [3] have developed an algorithm that accelerates the cell traversal process by sorting the elements by their maximum and minimum values to form two separate lists. They also calculate a global maximum, Δz, among the differences of all cells' two extreme values. When an iso-value S is initially specified or has changed from its previous value by more than Δz, all the cells with minimum values between $[S - \Delta z, S]$ are placed into an "active list." The active list is then purged of all cells that do not cross the iso-value. If the iso-value has changed less than Δz and is greater than its previous value, then the cells in the minimum list that have minimum values between [old S, new S] are placed into the active list. If the iso-value has changed less than Δz and is smaller than its previous value, then the elements in the maximum list with maximum values between [new S, old S] are placed into the active list. The updated active list is then purged of all cells that do not cross the iso-value.

While it can accelerate the cell traversal process, Gile's and Haimes' algorithm has several disadvantages. First, Δz cannot be too large, otherwise the range of $[S - \Delta z, S]$ can include a very large number of cells. The algorithm then degenerates to the naive method. Unfortunately, the range of Δz is hard to control because any cell with a large gradient could increase the global maximum by a large amount. One often sees this phenomenon in boundary value problems solved using a finite element method. Second, the algorithm performs well only when the user changes the iso-value smoothly. Third, keeping the active list updated requires list insertions and purges that further slow the process if either of the two previous conditions occur.

Span Filter: Gallagher designed a span filter to optimize the performance of his iso-surface extraction algorithm [4]. Initially, the range of the scalar values in the field is subdivided into several subranges termed *buckets*. The number of buckets that a cell's scalar values cross is defined as the span length. Cells are then distributed into different span lists according to their span lengths. Within each span list, the cells are further grouped into different buckets based on their lower bounds. For a given iso-value, the algorithm examines each span list. Within each span list, buckets that have bounds at and lower than the iso-value, depending on the span length of the list, are retrieved, and the elements inside are visited.

Gallagher's algorithm allows the user to specify iso-values randomly without performance penalties. However, because the cells are divided into different groups statically, and because within each group there is no ordering between cells, the algorithm cannot exploit coherence when two given iso-values are very close to each other. Because the data range is not divided based on the field's scalar distribution, there may be cells clustering in only a few groups.

3 Algorithms

In this section, we present the details of our iso-surface extraction algorithm. The algorithm is divided into two main parts: a sweep algorithm and a data decomposition scheme. The sweep algorithm incrementally locates the surface-intersected cells with a minimal amount of cell traversal. The data decomposition scheme guards the algorithm's worst case performance. We describe the sweep algorithm first because it is the kernel of the process and is applied to each subdivided group. We then explain our data decomposition scheme. Finally, we describe the integration of both the techniques that make up the *Sweeping Simplices Algorithm.*

3.1 Sweep Algorithm

A cell's extreme values are defined as the maximum and minimum scalar values at the corners of the cell. Straightforwardly, only those cells that have lower minimum values and higher maximum values than a

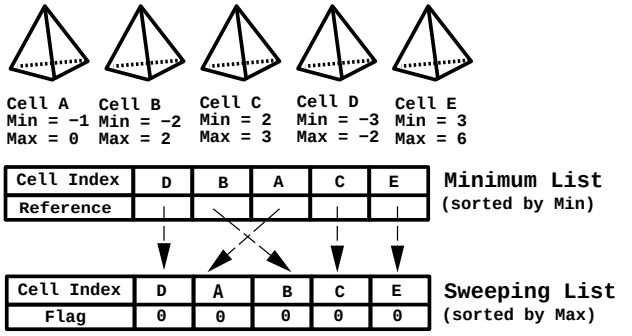

Figure 2: Minimum list and sweeping list after initialization.

given iso-value are intersected by an iso-surface. To efficiently and accurately locate candidate cells without searching the entire set of data, one must inevitably sort the field's cell elements by extreme values. To accomplish this sorting, the sweep algorithm maintains two lists, a *minimum list* and a *sweeping list*. Each cell in the field has a corresponding node in both lists. The minimum list is sorted by the cells' minimum values, and the sweeping list is sorted by the cells' maximum values. A node in the minimum list, termed *MinNode*, contains a represented cell's global index and a reference to that cell's corresponding node in the sweeping list. The corresponding node in the sweeping list, *SWNode*, contains the represented cell's global index and a flag, indicating whether the cell's minimum value is smaller than the current iso-value.

The algorithm starts by sorting the SWNodes in the sweeping list by their represented cells' maximum values. All SWNode flags are initially set to 0. To make cross-references from the minimum list to the sweeping list, we record each SWNode's address into its corresponding MinNode's reference field in the minimum list and then sort the minimum list by the represented cells' minimum values. Figure 2 illustrates the two lists for a scalar field containing five cell elements.

When an iso-value has been initially specified, a binary search algorithm is invoked to find the limit $I_{\min}$ in the minimum list where all the cells with MinNodes before $I_{\min}$ have lower minimum values than the iso-value. The algorithm then sweeps through those nodes before $I_{\min}$ and follows their references to the sweeping list to set the flags of the SWNodes. A limit, $I_{\max}$, is identified in the sweeping list so that all cells with SWNodes after that limit have higher maximum values than the iso-value.

It is then clear that all cells with SWNodes remaining in the sweeping list after $I_{\max}$ and having their flags set are intersected cells. In Figure 3, the shadowed node is the SWNode for the cell B, which is an iso-surface intersected cell.

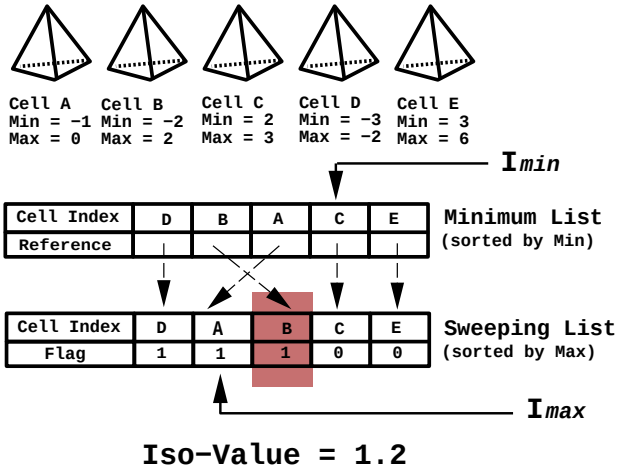

Figure 3: Sweep algorithm: Finding intersected cells.

If the iso-value is changed, the sweep algorithm locates a new $I_{\min}$ from the minimum list. Then all the MinNodes between the previous $I_{\min}$ and the new $I_{\min}$ are swept through and the references in the nodes are followed to their corresponding SWNodes in the sweeping list. If the new iso-value is larger than the previous iso-value, then those affected SWNodes' flags are set to 1 because their represented cells now have lower minimum values than the new iso-value. If the new iso-value is smaller than the previous iso-value, then those affected SWNodes' flags are reset to 0 because now their represented cells have higher minimum values than the new iso-value. A new $I_{\max}$ is found again and all the cells after that limit that have SWNodes with a flag set are the cells intersected by the iso-surfaces.

The algorithm is efficient because the minimum and sweeping lists can be updated incrementally. In addition, there is no list insertion or purge operation needed. Thus, although the algorithm has its best performance when the iso-value is changed smoothly, it still has very good performance even when the iso-value is changed abruptly. While some of the flags in the sweeping list must be examined sequentially for detecting non-intersected cells, we have found that sequential flag checking can be performed rapidly.

In general, the performance of the sweep algorithm is linearly proportional to the size of the sweeping list and also varies depending on the scalar distance between two consecutive iso-values. To avoid degenerative cases and to guarantee a good average performance when the user changes the iso-value randomly,

we have developed a data decomposition scheme that divides the cell elements in the entire field into several subgroups according to the ranges of scalar values within cells. The sweep algorithm is then applied only to a subset of the entire field for a given iso-value. In the following section we discuss the data decomposition algorithm and then explain the interaction of the sweep algorithm and the decomposition scheme.

3.2 Data Decomposition

The data decomposition scheme begins by subdividing the field's scalar range into subranges, which serve as the scalar boundaries to subdivide the cells into subgroups. A cell is assigned to a subgroup if both its minimum and maximum extremes are within the subgroup's scalar range. Obviously, the resultant subgroups can contain only cells that do not have boundary-crossing extreme values. We classify such subgroups as the bottom level, or level 0, in our data decomposition scheme. To include those cells crossing the boundaries, we combine the scalar ranges of every pair of even and odd numbered subgroups to form a new set of scalar boundaries for subgroups at one higher level, level 1. Of course, there might still be cells crossing the group boundaries at this level, so we need to follow the same rule to create higher levels of subdivision. This process continues until we arrive at the level in which only one group has the field's entire scalar range as its boundary. In Figure 4 cell A is put into group 0 at level 0. Cell B crosses the boundary of group 0 and group 1 at level 0, so it is in group 0 of level 1. Cell C crosses the boundary of group 2 and group 3 of level 1, so it is sent to group 1 of level 2. Cell D is in group 0 of level 3 because it crosses the group 0 and group 1 of level 2.

To select the initial scalar subranges, the algorithm first sorts the cells' maximum values into a list. It then selects the scalar ranges that can evenly subdivide the list as the scalar boundaries for the subgroups at level 0. We use the cells' maximum values to estimate and balance the size of each subgroup. For example, if 10% of cells have maximum values less than 0.1 and greater than 0.0, and a total of 20% of cells have maximum values less than 0.2, then there is a high probability that approximately 10% of cells are located in each of the ranges $[0.0, 0.1]$ and $[0.1, 0.2]$, assuming that the majority of the cells have smooth scalar transitions at their vertices.

If each subgroup is to have a corresponding subgroup to combine with at each level, the number of subgroups at the bottom level needs to be a power of two. Assuming that we have 2^N subgroups at the

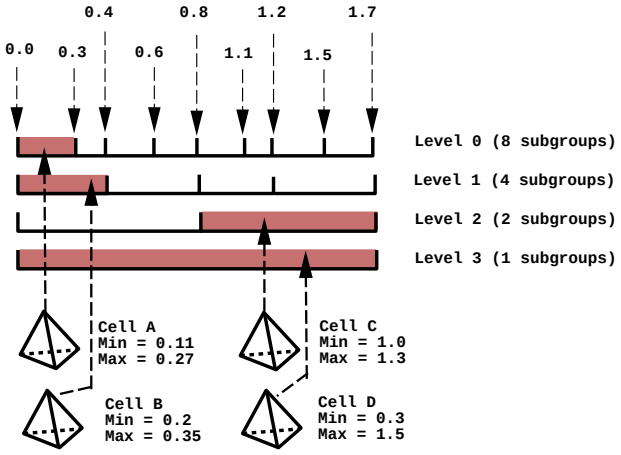

Figure 4: Data decomposition

level 0, then there will be a total of $N + 1$ levels, and the total number of subgroups will be $2^{N+1} - 1$. For a given iso-value, if this value is in the range of group G at level 0, then the value's corresponding subgroup at level K is $G/2^K$, assuming that the indices of subgroups at each level start from 0.

3.3 Sweeping Simplices Algorithm

After subdividing the entire set of cells into subgroups of different levels, we create first a sweeping list and then a corresponding minimum list for each subgroup by invoking the initialization method described previously. The initialization work of each subgroup can be performed right after the cells are distributed or can be delayed until the first time the group is selected for a given iso-value. The overall performance is the same either way because the initialization needs to be done only once. When a new iso-value is specified, one subgroup at each level is selected. The sweep algorithm is then invoked to find the intersected cells. Figure 5 contains pseudo code that illustrates the algorithm outline.

4 Algorithm Analysis and Simulation Results

We have implemented our algorithms in C++. The Open Inventor libraries [5] along with OpenGL [6] are used to facilitate three-dimensional viewing. In this section, we discuss the effectiveness of our data decomposition scheme, the performance of the sweep algorithm, and the respective memory allocations of each. We evaluate our algorithms using four sets of

```
for (int level=0; level<MAX_LEVEL; level++)
{
// find the target group at each level
group_id = find_group(level, iso-value);
// update minimum and sweeping lists
update_lists(group_id, iso-value);
// scan through the sweeping list
// and reconstruct triangular surfaces
find_triangle(group_id);
}
```

Figure 5: Pseudo code for the sweeping simplices algorithm.

Data Set	Size
Algebraic	34,295 Elements
Heart	69,892 Elements
Torso	397,010 Elements
Brain	471,770 Elements

Table 1: Data sizes

scalar data on unstructured grids. Three data sets consist of bioelectric field problems solved using the finite element method. The fourth data set is generated from an algebraic function. The data sets range in size from thirty thousand elements to four hundred thousand elements. Table 1 lists the size of each data set. All simulation results were performed on a 100 MHz MIPS R4400 processor.

4.1 Data Decomposition Scheme

The data decomposition scheme restricts the sweep algorithm's worst case performance by subdividing the elements in the field into different levels. The elements at each level are further subdivided into several subgroups. For a given iso-value, only one subgroup at each level needs to be investigated. The total size of the matched subgroup at each level, termed the *active size*, is the number of cell elements involved in the sweep algorithm at a give instance in time.

Figure 6 shows the ratio of active sizes to the size of the entire field. We experimented with different numbers of subdivisions at the bottom level to see how the average active size is affected. Each number presented here is an average result of ten thousand executions with randomly generated iso-values.

The figure shows that our data decomposition scheme effectively subdivides the elements into several subdomains. The active sizes become smaller as

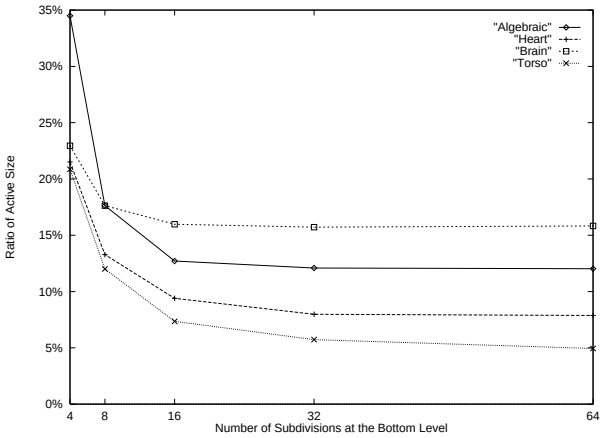

Figure 6: Ratio of average active size to the entire filed size for different numbers of subdivisions at the bottom level.

we increase the number of subdivisions at the bottom level. When the number of subdivisions exceeds a certain level, the active sizes converge to a stable state. This occurs because the scalar range of each subgroup becomes too narrow to subdivide the elements effectively and thus pushes them into subgroups at a higher level. The appropriate number of subdivisions at the bottom level is empirically determined and depends primarily on the scalar value distribution of the data set. Generally, we subdivide the field in such a way that the active size is approximately $1/8 - 1/20$ the size of the entire element set.

4.2 Sweep Algorithm

Initialization: In the initialization phase, the elements at each subgroup are sorted by their maximum and minimum values to form a minimum list and a sweeping list. In our implementation, we find the extreme value for each element when we input the scalar data and we use a quick-sort algorithm to sort the elements. Table 2 lists the CPU time for data input and maximum and minimum value sorting.

Iso-Surface Extraction: We divide the total iso-surface extraction time into cell traversal time and surface construction time. The cell traversal process includes all the necessary work to locate the iso-surface intersected cells. The surface construction process includes the operations of cell classification and interpolation on each of the intersected cells. The cell classification process checks the values at each of a cell's vertices and maps that cell into the lookup table. The interpolation locates the vertices of triangular surfaces. For the surface construction process, our

Data Set	Data Input	Sorting
Algebraic	1.30	5.67
Heart	2.70	12.79
Torso	15.89	73.99
Brain	22.93	109.07

Table 2: Initialization time: Data input and min/max sorting (in seconds).

Cell classification: a	5 μsec.
Interpolation: b	4 μsec.
Binary search constant: c	1 μsec.
Flag setting: d	0.5 μsec.
Flag checking: e	0.3 μsec.

Table 3: Execution time for different operations.

sweeping simplices algorithm spends the same time as the naive algorithm because we adopt the same divide-and-conquer paradigm to locate the surface within each cell. Assuming that there are N elements in the field, that the percentage of cells intersected by the iso-surfaces is f, that the cell classification time per cell is a, and that the interpolation time per cell is b, then the execution time of surface construction for a given iso-value is:

$$Nf(a+b)+C, \tag{1}$$

where C represents a constant time involved in the execution.

The cell traversal process in the sweep algorithm includes a constant time to find a matched subgroup at each level. Within each of those subgroups, binary searches for *Imin* and *Imax*, a sequential flag set, and a sequential flag checking operation are required. Assuming that there are L levels in the decomposition, that the average size for a subgroup is M, that the average time to set one flag is d, that the average time to check one flag is e, and that the binary search time for a group sized K is $c\log(K)$, then for a given iso-value, the average cell traversal time is:

$$L(2c\log(M)+d(M/2)+e(M/2))+C \tag{2}$$

In the second and third terms of the expression, we use $M/2$ as the average size of nodes whose flags need to be set and checked at a time. The C is a constant time involved in the program execution.

In the naive algorithm, the cell traversal time is the cell classification time spent on those non-intersected cells, which is:

$$N(1-f)a+C \tag{3}$$

To estimate constants a through e, we repeatedly performed a large number of executions and averaged the accumulation time. Table 3 lists our estimates for the constants. As for the average subgroup size M and number of levels L, we know from Figure 6 that if we cut our data domain into 6 levels (32 subgroups at the bottom level), the active size is approximately 10% of the total numbers of the cells. Therefore, if we assign L equal to 6, then the average size of a subgroup M is approximately $N \cdot 10\% \cdot (1/\mathrm{L}) = N/60$.

Substituting these constants into the performance expressions above and ignoring the constant C, we find that the naive algorithm's execution time for an iso-value is $9Nf+5N(1-f)$. The execution time for the sweep algorithm is $9Nf+12\log(1/60N)+1/25N$. The sweep algorithm spends the same time, $9Nf$ μseconds, as the naive algorithm to construct the surfaces within the surfaces-intersected cells. However, when we compare the $5N(1-f)$ μseconds in cell traversal time of the naive algorithm with the sweep algorithm's $12\log(1/60N)+1/25N$ for additional processing, we see a significant advantage. When N is large, the logarithmic term can be ignored. Unless the percentage of non-intersected cells is lower than 1%, the sweep algorithm will always outperform the naive algorithm.

Table 4 lists the average execution times for the naive algorithm and for the sweeping simplices algorithm. To demonstrate the incremental ability of our algorithm, we smoothly change the iso-value. In Table 5, we randomly select the iso-value to show our algorithm's average performance. Table 6 lists the savings of our algorithm for smoothly and randomly changed iso-values. Figures 7–10 depict a single iso-surfaced image for each of the data sets.

4.3 Memory Consumption

Assuming that there are M grid nodes and N tetrahedral elements in the field, the minimum space required to create a scalar field of unstructured mesh is $4N+4M$. This includes $3M$ words to store the grid nodes' $[x,y,z]$ positions, $4N$ words to store the connectivity information for each tetrahedron, and an additional M words to store the scalar data on each grid point. We use *word* as a memory unit to avoid involving implementation details.

Besides the minimum space $4N+4M$ required to accommodate unstructured grid data, our sweeping

	Naive algorithm		Sweep algorithm	
	Cell trav.	Surf. con.	Cell trav.	Surf. con.
Algeb.	134	73	1	73
Heart	307	123	2	123
Torso	1964	405	8	405
Brain	2332	455	18	455

Table 4: Execution time: Iso-value smoothly changed (in milliseconds).

	Naive algorithm		Sweep algorithm	
	Cell trav.	Surf. con.	Cell trav.	Surf. con.
Algeb.	132	81	9	81
Heart	331	131	14	131
Torso	1958	415	48	415
Brain	2367	481	66	481

Table 5: Execution time: Iso-value randomly changed (in milliseconds).

simplices algorithm needs an additional $2N$ words to store the maximum and minimum values for each cell element, $2N$ words for storing the cell's indices in the nodes of minimum lists and sweeping lists, N words to store the pointers from nodes of minimum lists to sweeping lists, and N bits assigned to the flags in the sweeping list. Therefore, the additional space required by the algorithm sums to $5N$ words $+1Nbits$. Considering that N is typically 4 to 5 times as large as M, the sweeping simplices algorithm requires about twice as much space as the minimum memory required to represent a scalar field on an unstructured grid.

5 Conclusion and Summary

We have presented a fast iso-surface extraction algorithm for data on unstructured grids. The algorithm comprises two parts, a sweep algorithm and a data decomposition scheme. The sweep algorithm can incrementally locate the cells that are intersected by iso-surfaces while the data decomposition scheme can restrict the algorithm's worst case performance by decomposing the data elements into subdomains.

Currently, we are in the process of designing a better data structure that utilizes spatial coherence between iso-surfaces so we can reduce interpolation costs. In addition, we are developing a parallel version of our algorithm.

Acknowledgments

This work was supported in part by awards from the NIH and NSF. The authors would like to thank J. Schmidt and D. Weinstein for the biomedical simulation data. We would also like to thank C. Hansen, Y. Livnat and K. Coles for their helpful comments and suggestions. Furthermore, we appreciate access to facilities which are part of the NSF STC for Computer Graphics and Scientific Visualization.

	Sweep algorithm savings			
	Smoothly		Randomly	
	Cell Trav.	Total	Cell Trav.	Total
Algebraic	99%	64.1%	93.4%	58.0%
Heart	99%	70.9%	95%	68.7%
Torso	99.5%	82.6%	97.5%	80.5%
Brain	99.2%	83.0%	97.2%	80.8%

Table 6: Execution savings: Iso-value smoothly and randomly changed.

References

[1] W.E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[2] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[3] M. Giles and R. Haimes. Advanced interactive visualization for CFD. *Computing Systems in Engineering*, 1(1):51–62, 1990.

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

[5] J. Wernecke. *The Inventor Mentor*. Addison-Wesley Publishing Company, 1994.

[6] J. Neider, T. Davis and M. Woo. *OpenGL Programming Guide*. Addison-Wesley Publishing Company, 1993.

6 Figure Captions

Figure 6. An iso-surface within an algebraic scalar field.

Figure 7. Iso-surfaces of constant voltage from a finite element simulation of cardiac defibrillation within the ventricles of the human heart.

Figure 8. An iso-surface 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.

Figure 9. An iso-surface of constant voltage from a finite element simulation of temporal lobe epilepsy in a model of the human skull and brain.