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 isovalue 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. The algorithm uses a divide-and-conquer scheme to locate the surface within each cell that serves as a logical unit encompassed by eight data values at its corners. The algorithm scans through each cell, analyzes the data, and then maps the cell into a lookup table. The lookup table provides the necessary information for fast iso-surface reconstructions. The algorithm can be further accelerated by using pixel-to-pixel, line-to-line, and slice-to-slice coherence [1], and by adopting spatial decomposition methods such as octrees [2].

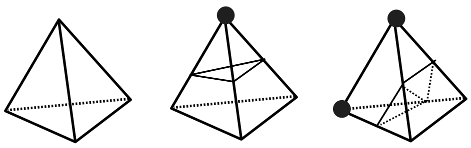

Figure 1: Surface-Cell Intersection Geometry

Although the marching cubes algorithm originates from medical applications, which typically use structured grids, the algorithm can be extended to problems using unstructured grids or 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 with tetrahedral elements, there are only three different types of surface-cell intersection geometry, as shown in Figure 1. Although the naive algorithm is straigthforward to implement, it is very 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 $O(N^{2/3})$.

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

2.1 Octrees

Wilhelms and Van Gelder [2] use an octree data structure to accelerate iso-surface extraction. The nodes of the octrees store the summary information about the underlying sub-volumes. This summary information can be a flag to indicate whether an entire subvolume is empty, can be the maximum and/or minimum scalar value among all the encompassed elements, or can be other useful information. Wilhelms and Van Gelder's algorithm starts with a setup phase that creates the octrees. 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 skipped. 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, octrees data structures are primarily implemented for data on structured grids. Moreover, octrees method is sensitive to high-frequency noise within the data.

2.2 MinMax Lists

It is obvious that only the cells that have higher maximum value and lower minimum value than the given iso-value are intersected by the iso-surfaces. 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 "action list." The action 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 put into the action 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 action list. The updated action 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 action list updated requires list insertions and purges which further slow the process if either of the two previous conditions occur.

2.3 Span Filter

Gallengher 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.

Gallengher'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 utilize 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 could 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 first describe the sweep algorithm 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 given iso-value are intersected by the iso-surfaces. 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. A node in the minimum list, termed *MinNode*, contains its represented cell's global index and a pointer to that cell's corresponding node in the sweeping list. The node in the sweeping list, *SWNode*, contains its represented cell's global index and a flag indicating whether the cell's minimum value is smaller than the current iso-value.

The sweep algorithm is performed in two phases: an initialization phase and a dynamic phase. In the initialization phase, the minimum list and sweeping list are sorted and links between them are created. 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 links between the minimum list and the sweeping list, we record each SWNode's address into its corresponding MinNode in the minimum list and then sort the minimum list by the represented cells' minimum values. Figure 2 illustrates the two lists after initialization.

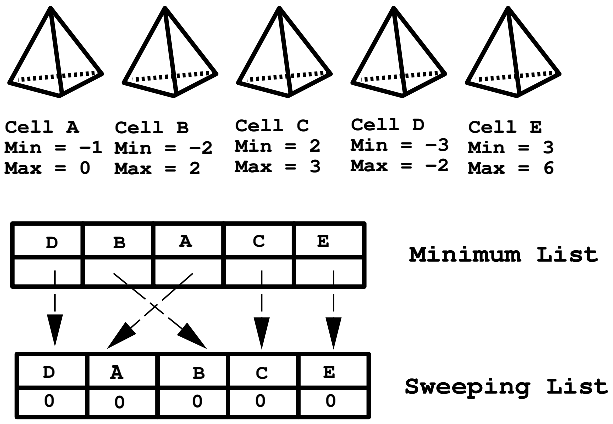

Figure 2: Minimum List and Sweeping List After Initialization

When an iso-value has been initially specified, the sweep algorithm begins the dynamic phase. 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 pointers 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 flag set are intersected cells. In Figure 3, the shadowed node is the SWNode for the cell B, which is an iso-surface intersected cell.

If the iso-value is changed, the sweep algorithm updates the sweeping list. First, a new $I_{\min}$ is found from the minimum list. Then all the MinNodes between the previous $I_{\min}$ and the current $I_{\min}$ are swept through and the pointers in the nodes are followed to their corresponding SWNodes in the sweeping list. If the previous $I_{\min}$ is smaller than the current $I_{\min}$, 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 previous $I_{\min}$ is larger than the current $I_{\min}$, 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 sweeping list

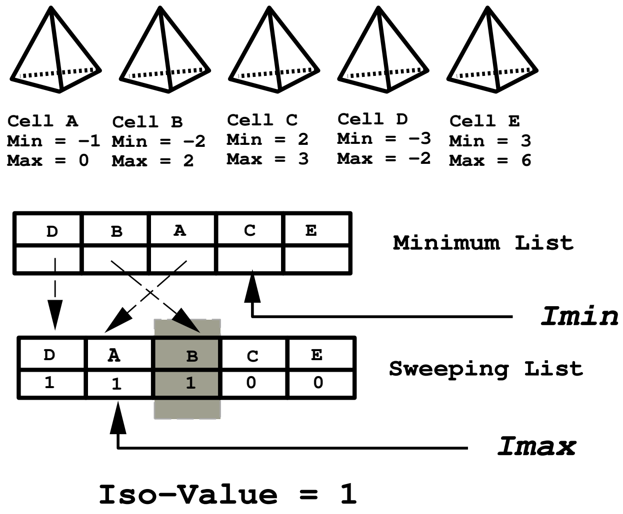

Figure 3: Sweep Algorithm: Dynamic Phase

can be rapidly updated without any node insertion or purge operations. In addition, since the algorithm stores the previous iso-value and its corresponding $I_{\min}$ and $I_{\max}$, lists of those values can be updated incrementally. Thus, the algorithm has its best performance when the iso-value is changed smoothly, but 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 very quickly.

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 of 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 subdivision criteria and algorithm and then explain the interaction of the sweep algorithm and the decomposition scheme.

3.2 Data Decomposition

There are two design criteria for our data decomposition scheme. First, for a particular iso-value, the subgroups that have a probability of containing the intersected cells must be rapidly determined. Second, the sizes among the subgroups need to be kept as balanced as possible. Our algorithm satisfies the first criterion by subdividing the cells based on their extreme values. To satisfy the second criterion, our algorithm takes the entire field's scalar distribution into account.

The data decomposition scheme begins by selecting scalar ranges that can subdivide the cells into several subgroups. It achieves this by sorting the cells' maximum values into a list. The scalar values that evenly subdivide the list are then found. These scalar values constitute the scalar ranges, or boundaries, for the subgroups. We use the cell's maximum values to estimate 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.

A cell is classified to a subgroup only when the scalar values at its vertices are within the subgroup's scalar range. Obviously, the above boundaries are not enough to completely and uniquely partition the cells. Those subgroups can only contain cells that do not have boundary-crossing extreme values. We define these subgroup's level as the bottom level, level 0, of our decomposition scheme. For those cells crossing the boundaries at this level, we combine the boundaries of every pair of even and odd numbered subgroups to form a new subgroup's scalar boundary at another 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.

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 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.

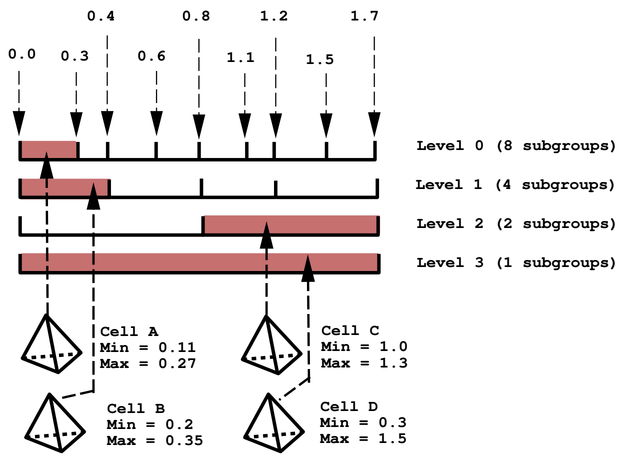

Figure 4: Data Decomposition

For a given scalar value, there is only one subgroup at each level whose range contains that scalar value. We now distribute the cells as follows. For each cell, we find two of the groups at level 0 whose ranges contains that cell's maximum and minimum values. If these two groups are different, we know that this cell's extreme values cross the group boundaries. We then locate the groups containing that cell's extreme values at one higher level. We continue the process until we arrive at a level that has a group containing both the cell's extreme values. We then send the cell into that group. This cell classification process is very fast because once we know the group indices for a cell's extreme values at level 0, the group indices at other levels can be calculated by the relation previously described.

3.3 Combination of Data Decomposition and Sweep Algorithm

The integration of the data decomposition scheme and the sweep algorithm is straightforward. After subdividing the entire set of cells into several subgroups, 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 dynamic algorithm described is then invoked to find the intersected cells.

Following is a piece of pseudo code which illustrates the algorithm outline.

```
for (int level=0; level <MAX_LEVEL; level++)
{

// find the target group at the current 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);

}
```

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

Table 1: Data Sizes

4 Algorithm Analysis and Simulation Results

We have implemented our algorithms in C++. The Open Inventor libraries along with OpenGL 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 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

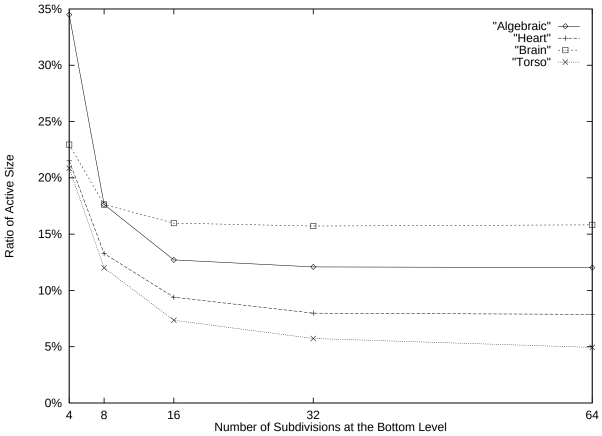

Figure 5: Ratio of average active size to the entire filed size for different number of subdivisions at the bottom level

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. Since the sweep algorithm's performance is linearly proportional to the size of the subgroups, we wish to minimize the size of each subgroup.

Since we need to know the average size of elements that must be processed for a given iso-value under our data decomposition scheme, we randomly generate an iso-value and then sum up the size of the matched subgroup at each level. The resultant size, termed *Active Size*, is the number of elements involved in the sweep algorithm at a given instance in time. Figure 5 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 repeated executions.

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

Data Set	Data Input	Sorting
Algebraic	1.30	5.673
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)

entire element set.

4.2 Sweep Algorithm

4.2.1 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.

4.2.2 Dynamic Phase

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 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, 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

Cell Classification: a	5μ sec.
Interpolation: b	4μ sec.
Binary Search Constant: c	1μ sec.
Flag Setting: d	0.5μ sec.
Flag Checking: e	0.3μ sec.

Table 3: Execution time for different operations

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:

$$L(2c\log(M) + d(M/2) + e(M/2)) + C \qquad (2)$$

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. It is:

$$N(1-f)a + C \qquad (3)$$

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, from Figure 5, we know 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, $9Nf\mu$seconds, as the naive algorithm to construct the surfaces within the surfaces-intersected cells. However, when we compare the $5N(1-f)\mu$seconds 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 and Table 5 list the average execution times and savings for the naive algorithm and our sweeping simplices algorithm. We smoothly change the iso-value to demonstrate the incremental ability of our algorithm. In Table 6 and Table 7, we randomly select the iso-value to show our algorithm's average performance. Figures 6–9 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 additional M words to store the scalar data on each grid point. We use *word* as a memory unit to avoid involving implementation details. For certain applications, it is necessary to create more complicated data structures that include face-to-face or node-to-node neighbor information. These data structures would require a much larger memory space.

Besides the minimum space $4N + 4M$ required to accommodate unstructured grid data, our sweeping 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 $5N words + 1N bits$. 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 an unstructured grid scalar field.

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. Our algorithm uses a table lookup method, as in the marching cubes algorithm, to construct the triangular surfaces, but can substantially reduce the cell traversal time.

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

Acknowledgments

This work was supported in part by the Whitaker Foundation. 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.

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.

	Naive Algorithm		Sweep Algorithm	
	Cell Trav.	Surf. Con.	Cell Trav.	Surf. Con.
Algebraic	133.87	73.01	1.3	73.01
Heart	307.03	122.84	2.2	122.84
Torso	1963.94	404.83	8.21	404.83
Brain	2332.32	455.18	17.8	455.18

Table 4: Execution Time: Iso-Value smoothly changed (in $m$sec.).

	Sweep Algorithm	
	Cell Trav. Savings	Total Savings
Algebraic	99%	64.1%
Heart	99%	70.9%
Torso	99.5%	82.6%
Brain	99.2%	83.03%

Table 5: Execution Savings: Iso-Value smoothly changed.

	Naive Algorithm		Sweep Algorithm	
	Cell Trav.	Surf. Con.	Cell Trav.	Surf. Con.
Algebraic	132.01	80.5	8.75	80.5
Heart	331.19	130.56	14.11	130.56
Torso	1958.18	415.43	48.11	415.43
Brain	2366.52	480.54	66.36	480.54

Table 6: Execution Time: Iso-Value randomly changed (in $m$sec.).

	Sweep Algorithm	
	Cell Trav. Savings	Total Savings
Algebraic	93.37%	58.01%
Heart	95%	68.67%
Torso	97.54%	80.47%
Brain	97.19%	80.79%

Table 7: Execution Savings: Iso-Value randomly changed.

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 simuation 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.