12-2003

Isosurface Extraction in the Visualization Toolkit Using the Extrema Skeleton Algorithm

Subha Parvathy Mahaadevan
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Subha Parvathy Mahaadevan entitled "Isosurface Extraction in the Visualization Toolkit Using the Extrema Skeleton Algorithm." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Bruce A. Whitehead, Major Professor

We have read this thesis and recommend its acceptance:

Kenneth R. Kimble, L. Montgomery Smi

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Subha Parvathy Mahaadevan entitled "Isosurface Extraction in the Visualization Toolkit Using the Extrema Skeleton Algorithm." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Bruce A. Whitehead

Major Professor

We have read this thesis and
recommend its acceptance:

Kenneth R. Kimble

L. Montgomery Smith

Acceptance for the Council:

Anne Mayhew

Vice Provost and
Dean of Graduate Studies

(Original signatures are on file with official student records.)

ISOSURFACE EXTRACTION IN THE VISUALIZATION TOOLKIT USING THE EXTREMA SKELETON ALGORITHM

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Subha Parvathy Mahaadevan

December 2003

ACKNOWLEDGEMENTS

I wish to thank my advisor Dr. Bruce Whitehead for his guidance and support during my years of graduate study at UTSI. I also thank my thesis committee members Dr. Kenneth Kimble, and Dr. Montgomery Smith for taking the time to serve on my committee and for their valuable advice and input. My most sincere thanks are due to Dr. John Steinhoff and Dr. John Caruthers for giving me the opportunity to work with the CMRG team and for providing financial support for over a year, to Dr. Mary Helen McCay, Dr. Dennis Keefer, and James O. Hornkohl for helping me to continue and complete my thesis while working for CLA. Finally, thanks to Abraham Meganathan for his help and support during my thesis and my stay at UTSI.

ABSTRACT

Generating isosurfaces is a very useful technique in data visualization for understanding the distribution of scalar data. Often, when the size of the data set is really large, as in the case with data produced by medical imaging applications, engineering simulations or geographic information systems applications, the use of traditional methods like marching cubes makes repeated generation of isosurfaces a very time consuming task. This thesis investigated the use of the Extrema Skeleton algorithm to speed up repeated isosurface generation in the visualization package, Visualization Toolkit (VTK). The objective was to reduce the number of non-isosurface cells visited to generate isosurfaces, and to compare the Extrema Skeleton method with the Marching Cubes method by monitoring parameters like time taken for the isosurfacing process and number of cells visited. The results of this investigation showed that the Extrema Skeleton method was faster for most of the datasets tested. For simple datasets with less than 10% isosurface cells and complex datasets with less than 5% isosurface cells, the Extrema Skeleton method was found to be significantly faster than the Marching Cubes method. The time gained by the Extrema Skeleton method for datasets with greater than 15% isosurface cells was found to be insignificant. Based on the results of this study, implementing the Extrema Skeleton method for the VTK software is a change worth making because typical VTK users deal with datasets for which the Extrema Skeleton method is significantly faster and also with datasets for which it is marginally faster than the Marching Cubes method.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

Ω	A domain in $\mathcal{R}^3$
Σ	A grid
C_{-1}	A cell that has an extremum point as one of its vertices
C_i	A cell with 'i' neighbors (for $i \geq 0$)
e_i	A cell in a grid
f	A function
FIFO	First In First Out queue
k	Number of isosurface cells
K	Number of buckets
m	Number of extremum points in a dataset
n	Number of cells in a dataset
q	A fixed value in $\mathcal{R}$ for which an isosurface has to be generated
r	Number of triangles in a triangle strip
S	A set of all isosurface points corresponding to a fixed value in a domain
V	Finite set of points spanning a domain
v_i	Elements of set V
v	Vertex of a cell
W	Set of values of a scalar field sampled at points of V
w_i	Elements of set W

CHAPTER 1

INTRODUCTION

Extracting isosurfaces is one of the most widely used and effective techniques for visualizing scientific data. It is an effective technique for understanding the distribution of scalar fields in a three-dimensional data set.

An isosurface can be defined as a constant density function on a 3D data set. More precisely, a scalar volume data set [13] is a pair (V, W), where $V = \{v_i \in \Re^3, i = 1,\ldots,n\}$ is a finite set of points comprising a domain $\Omega \subset \Re^3$, and $W = \{w_i \in \Re, i = 1,..,n\}$ is a corresponding set values of a scalar field $f(x,y,z)$, sampled at the points of V, i.e., $w_i = f(v_i)$. Given a value $q \in \Re$, the set $S(q) = \{p \in \Omega \mid f(p) = q\}$ is called the *isosurface* of field f corresponding to the value q.

Often it is useful to study the distribution of scalar fields by repeated generation of isosurfaces corresponding to different scalar values q. This feature is supported in most visualization tools and applications. Most physical real world problems generate such huge amounts of data that the cost of generating an isosurface becomes high in terms of time and resources. Efficient algorithms are therefore necessary for fast and accurate isosurface generation.

The representation of the data set in 3D space can be summarized as follows [13]. To begin with, a grid Σ is given. Σ is formed by subdividing the domain Ω into small hexahedra or tetrahedra, termed 'cells', whose vertices are at the points of V and whose values are in W. Thus each point on the grid Σ is the pair (v_i, w_i), $i=1,..,n$. Consider a cell $e_j \in \Sigma$, that has vertices $v_{j1},.., v_{jk}$. e_j is called an 'isosurface cell' if, for a fixed $q \in \Re$, $\min_i \{w_{ji}\} \leq q \leq max_i \{w_{ji}\}$.

The process of generating an isosurface occurs in four stages -

(1) Finding isosurface cells: Given a value q, Σ is searched for isosurface cells.

(2) Determining the topology of the surface inside each isosurface cell: Based on the cell topology, there are only a specific number of ways in which a surface can pass through a cell. This is discussed in the Marching cubes method [1] in Chapter 2.

(3) Vertex computation: This is done by linear interpolation of the field along the edges that the isosurface intersects. (These vertices, called isosurface vertices, will be joined to form triangles at a later stage. Thus the isosurface will be approximated as a set of triangles inside each isosurface cell).

(4) Surface normal computation: For each vertex so formed, the normal to the surface at that point is calculated.

1.1 RELATED WORK

Techniques like the Marching Cubes algorithm, that visits every cell in a dataset to find isosurface cells, can be very time consuming when the dataset is large. Many techniques have been developed that aim to reduce the time taken by the cell selection stage. These techniques can be grouped as algorithms that sort cells according to their scalar values, algorithms that use space-subdivision for cell classification like the Octrees method [4], and algorithms based on seed set and range based approaches. The first two techniques work best for a structured volume and are difficult to apply to unstructured grids. Also, the number of cells visited is *O(n)* [4,5], where n is the total number of cells. However, algorithms based on seed set approaches and range-based approaches can be applied to both structured and unstructured grids. The computation time for the isosurfacing process for these algorithms is much less than *O(n)* [6,9].

The range-based methods use an interval [*a,b*] of a cell's scalar values, '*a*' being the cell's minimum scalar and '*b*' being the cell's maximum scalar. The isosurface cells are located by looking for an interval where $a \leq q \leq b$ where q is the desired isovalue. Algorithms based on this approach include K-d trees [9], Lattice classification [6], and Interval trees [13]. The number of cells visited in these algorithms is less than *O(n)*, but they require over *O(n)* time for the construction of the data structures in the initialization stage. The second approach is the seed set based method [7,8,10]. In this method, the preprocessing step generates a seed set, which is a set of cells in a volume chosen based on certain conditions. By traversing the seed set, it is possible to find at least one cell that

belongs to a desired isosurface. Using a propagation algorithm [8] that recursively visits adjacent cells (cells that share a face with a given cell), the entire isosurface is generated. Algorithms developed on the seed set based approach include the Extrema Graph method [8], and the Contour Tree method [7].

The Extrema Graph method extracts local maxima and minima, and connects them by arcs of cells in the volume. It then generates two sorted cell lists based on the maximum and minimum values of the cells on the boundary. This is done so that disjoint parts of an isosurface can be located. When a value is given, cells in the Extrema Graph and the boundary cell lists are searched to find at least one isosurface cell. The rest of the isosurface is propagated from these cell(s). The disadvantage with this method is having to maintain the boundary cell list. Also, complexity of the algorithm is dependent on the number of extremum points in the data set [8].

The Contour Tree method overcomes the need to store boundary cells by finding not only extrema points, but also saddle points, and connects them all using a tree structure. The preprocessing time for this method is more than $O(n)$ [7].

The Extrema Graph method was later extended to develop the Extrema Skeleton method [12]. In this method, the extremum points in a volume are found as in the Extrema Graph method, but instead of connecting the extrema points by a graph, a technique called Volume Thinning is used. The Volume Thinning technique is an

extension of the method used for image recognition. The Extrema Skeleton generated serves as the seed set. The Extrema Skeleton aims at preserving the topology of a given volume while connecting all the extrema in the volume. The skeleton contains at least one cell for every isosurface in the volume. The Extrema Skeleton method does not require more than $O(n)$ computation time for preprocessing since the preprocessing technique involves visiting cells a constant number of times [12]. The number of cells in the skeleton is estimated as $O(n^{1/3} m)$ [12], where m is the number of extremum points. The time for finding the isosurface cells is estimated as $O(n^{1/3} m + k)$ [12] , where k is the number of isosurface cells.

1.2 PROBLEM STATEMENT

Many modern day applications generate large quantities of data. Therefore, it is worth exploring techniques that reduce the time taken for finding isosurface cells. The VTK package uses the Marching Cubes algorithm to find isosurfaces. So, it was hypothesized that substituting the Marching Cubes algorithm with a more efficient algorithm would provide an improvement in the processing time. To test this hypothesis, the Extrema Skeleton algorithm was chosen. Various reasons shown below contributed to this choice :

- Compared to the Marching Cubes algorithm, the number of cells searched by the Extrema Skeleton algorithm is smaller.
- Preprocessing time is smaller than most methods that use sorting techniques.
- This algorithm can be applied to both structured and unstructured grids.

This thesis is an investigation of the hypothesis that the Extrema Skeleton algorithm will decrease the time taken for isosurface extraction in a given simple structured data set, compared to the Marching Cubes algorithm used by the VTK package.

1.3 OBJECTIVES

This thesis focuses on the following objectives to investigate the hypothesis described in the problem statement of Section 1.2 :

- Implement the Extrema Skeleton algorithm for the graphics package Visualization Toolkit (VTK) for simple structured grids consisting of voxels.
- Determine the time and number of cells visited for multiple isosurfaces generated in the same volume and compare it with the existing method in the package for finding isosurfaces viz., the Marching Cubes method.

Results of this study indicated that the Extrema Skeleton method was faster than Marching Cubes for most of the datasets tested. While for some datasets, the time improvement was significant, for others the time gained was negligible. It was concluded that implementing the Extrema Skeleton method for VTK speeds up the software by at least 10% for simple datasets with less than 10% isosurface cells. Detailed results and discussion are presented in Chapter 4.

1.4 APPLICATIONS

Continuous extraction of isosurfaces is particularly useful in applications where the distribution of scalar values is important. Examples of such applications include

visualizing distribution of temperature or pressure in fluid flow, and visualizing Magnetic Resonance (MR) scan data for extracting images of skin, soft tissues at various depths, bone structure, and organs from a single set of scanned data.

CHAPTER 2

LITERATURE REVIEW

The simplest way to compute isosurfaces is to visit each cell and determine whether the surface of interest passes through it. The isosurface is then approximated as a set of points or triangles inside the cells, as described in the Marching Cubes method [1]. Since this requires that all the cells in a given data set be visited, the time required for it is always *O(n)* [1], where *n* is the number of cells in the data set. This would be a disadvantage for particularly large values of *n*, since not all cells are isosurface cells. Hence, different approaches aimed at reducing the number of non-isosurface cells visited were proposed by different researchers. Sections 2.2 through 2.6 discuss some of the algorithms that were developed on this idea. The algorithms discussed below are the Marching Cubes method [1], Span filtering [5], Octrees [4], Issue algorithm [6], Contour trees [7], and Extrema graph method [8] .

2.1 MARCHING CUBES

The Marching Cubes algorithm [1] creates triangle models of constant density surfaces from 3D data. The algorithm deals with surface reconstruction and involves the creation of a surface model from 3D data. The model usually consists of 3D volume elements (voxels) or polygons. A certain density value corresponding to the surface that is desired to be visualized is specified. Surface reconstruction then takes place in two stages. The algorithm first locates the surface corresponding to the specified value and

creates triangles. Then, it creates normals to the surface at each vertex of the triangle to construct a visualizable image.

This method addresses all four processing stages for isosurface extraction listed in Chapter 1. The first two steps, finding the isosurface cells and determining the topology of the surface, are performed in the same pass.

2.1.1 Finding Isosurface Cells and Determining the Topology of the Surface

The algorithm determines how a surface intersects a logical cube, created from eight pixels, and then moves on (marches) to the next cube until the whole image is constructed. To find how a surface intersects a cube, the algorithm assigns a "1" to all vertices in the cube for which the data value at the vertex exceeds or equals the value of the surface. If not, it assigns a value "0". The vertices with value '1" are considered inside or on the isosurface while the vertices with value "0" are outside the surface of interest. The surface intersects all those cube edges such that one vertex is outside and another is inside the surface. Based on this, the topology and the location of the surface inside the cube are determined. Since a regular cube has eight vertices, and the state of each vertex can either be inside or outside a given surface, there are $2^8 = 256$ cases for surface-edge intersection. For the purpose of triangulation, the number of cases is reduced by the application of two different symmetries as described by Lorensen [1]. The cases resulting from the two different symmetries are the same and so finally, the total number of cases is reduced to 14. One of the symmetries occurs since the topology of the

surface inside the cube is unchanged if the relationship of the surface value to the “0” and “1” value of the vertices is interchanged. The vertices with value “1” are changed to value “0” and vice versa. Thus, only cases with zero to four vertices whose values are greater than the surface value are taken into consideration. This reduces the number of cases to 128. The number of cases is further reduced to 14 by applying rotational symmetry. Figure 2.1 shows an illustration of the 14 basic cases.

2.1.2 Vertex Computation and Surface Normal Computation

Vertex computation and surface normal computation are the two final steps of the isosurface extraction process. An eight-bit index is created for each case based on the state of each of the eight vertices. This index is then used as a pointer to look up the appropriate list of intersected edges for the respective case in the table of surface-edge intersections. Using this entry, the surface intersection along the edge is interpolated. Finally, normals are created at the resulting triangle vertices to produce Gouraud-shaded images.

Marching cubes is a simple, yet a very powerful algorithm and is used in many applications even today. The algorithm visits every cell in the data set. The complexity of the algorithm is $O(n)$ since every cell is visited once [1]. This might become a disadvantage in the case of a large data set. To find the isosurface corresponding to a new value, the whole data set is visited once again. There is no stored information from a

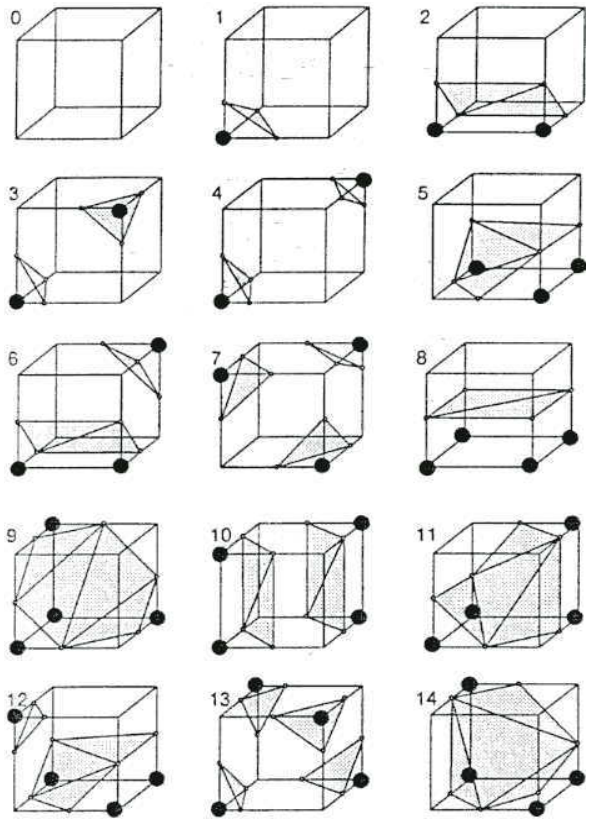


Figure 2.1: Triangulated Cubes (Image courtesy of Kitware, Inc. and taken from [1])

previously generated isosurface, and there is no carry over information from the previous iteration.

2.2 SPAN FILTERING

The Span Filtering method, proposed by Gallagher [5], focuses on the first stage in the isosurface extraction process, i.e., finding isosurface cells. Any polygonization algorithm could be used in conjunction with this search method, to achieve the same end result as the final three stages of the isosurface extraction process listed in Chapter 1.

This algorithm classifies or sorts cells according to the values in a cell and generates a compressed data representation for speeding up isosurface generation. The range of data values is divided into sub ranges, termed “buckets”. Every cell is classified based on which bucket its minimum value falls in and the number of buckets the particular cell’s range spans. In a “span list” cells are grouped based on their span, and within each group, cells are grouped further according to their starting bucket value. There is a list for groups that belongs totally to one bucket, one for groups that span more than K buckets, and one for groups whose span is greater than the previous lists. The search algorithm for this method has a complexity of $O(n)$ [5].

2.3 OCTREES

As in the case of the Span Filtering method, the Octrees algorithm, proposed by Wilhelms [4], focuses only on the cell selection stage of the isosurface extraction process.

While Wilhelms used the Octrees algorithm for finding the isosurface cells, the polygonization of those isosurface cells was performed using the Marching Cubes subroutines (determining the topology of the surface, vertex computation, and surface normal computation), discussed in Section 2.1.

Octrees are hierarchical data structures based on the decomposition of three-dimensional space, that recursively divide 3D space into eight sub-volumes [4]. The root of the octree refers to the entire volume. Every coordinate direction is divided into a "lower" half space and an "upper" half space to create octants. A volume that has 2^{n-1} and 2^n cells can be represented by an octree of depth '*n*'. Wilhelms [4] discusses the usage of summary at each node of the octree for the entire subvolume beneath it for isosurface generation. Thus, only areas of the tree that correspond to the value of interest need to be explored. For isosurface extraction, the maximum and minimum values of the data within a node's sub-region are maintained. Given a value for which the corresponding isosurface has to be extracted, only those nodes whose minimum values are not greater than the given value and whose maximum values are not less than the given value are traversed. Nodes for which this condition fails, and the entire branch below them, are not traversed.

The number of cells visited in this method in general is twice the number of isosurface cells [4]. The tree is sensitive to the underlying data it represents and so if the data contains many fluctuations or noise, then most of the tree needs to be traversed.

Also, the method requires significant setup time. So, the pre-computed results have to be stored effectively for any speed advantages.

2.4 ISOSURFACE PROPAGATION

The Isosurface Propagation algorithm proposed by Speray and Kennon [3] combines the cell selection stage and the polygonization step. This method does not make clear distinctions between the surface topology determination, vertex computation, and surface normal computation steps. Instead, polygons are generated as and when isosurface cells are found.

The propagation algorithm does not require much preprocessing. It uses cell adjacency to propagate itself through the face of cells. It requires a data structure that maintains the IDs of adjacent cells for each cell. When a cell that is intersected by an isosurface is specified, all its adjacent cells are put into a queue. Each cell is then extracted from the queue to check whether the surface of interest passes through it. If so, the adjacent cells of the newly found isosurface cell are added to the queue. This process is done until the queue becomes empty. The algorithm requires manual specification of the starting cell and is not very useful for generating isosurfaces without it.

2.5 EXTREMA GRAPH

The Extrema Graph method [8] focuses on the cell selection stage of the isosurface extraction process. The Marching Cubes subroutines or any other

polygonization algorithm could be used in conjunction with the Extrema Graph method to generate the isosurface.

This method is based on generating a small seed set for a given data set. In the seed set based approach, a small set of cells from the data set is first generated. Given a fixed isovalue, the algorithm searches the seed set for a "hit" cell, or cells that contain the surface of interest. Using these cells as the starting point, the adjacent cells of each of these cells, i.e., the cells that share a face with a specified cell, are searched recursively, as in a propagation algorithm described in Section 2.4. This is done until all the cells that contain the isosurface are found. A polygonization algorithm, like Marching Cubes [1], is used to create triangles for isosurface generation. The preprocessing phase of the algorithm examines the values at the vertex of every cube in the volume and generates a set of extrema[1] points. The algorithm makes the following assumptions [8]:

1) if the isosurface is closed, then there are extrema points both inside and outside it[2].
2) if an isosurface is open then it intersects the boundary of the volume.

[1] **Definition:** Let 'e_1' be a hexahedral cell in a given dataset and let $v_{11}, v_{12}, v_{13}, \ldots, v_{18}$, be the vertices of this cell. Let $e_2, e_3, e_4, e_5, e_6, e_7, e_8$ be the seven cells that have the vertex $v_{11}, v_{i2}, \ldots, v_{i8}$, where $i = 2,3\ldots,8$. In other words, they are connected to e_1 by 0 or more cell edges and the vertex v_{11} is a common vertex of all the eight cells. The vertex v_{11} is termed a minimum if $v_{11} < v_{ij}$, (or equivalently, a maximum if $v_{11} > v_{ij}$), for all $i,j = 1,2,3,..,8$ $(i,j \neq 1,1)$. If v_{11} is either a maximum or a minimum, then it is called an extremum point.

[2] A closed isosurface divides the dataset into two disjoint finite sets each of which must contain at least one extremum point.

2.5.1 Finding Extrema Points

Itoh [8] defines extrema points as grid points whose scalar values are maximum or minimum in *all* cells that share them. To find the extremum points, the scalar values at the vertices of each cell are compared with one another, beginning with the first cell in the dataset. Consider the case for finding minima - in each cell, the vertex *v* with the least scalar value is marked as the minima. It is unmarked later if the scalar value of *v* is not the minimum in another cell that contains *v* as one of its vertices. When all cells are processed in this manner, this will leave a few vertices marked as mimima. The same procedure is followed to find the maximas. A vertex that is marked either maxima or mimima is an extremum point.

2.5.2 Generating Extrema Graphs

An extrema graph is defined as a set of arcs that connects two extrema points. To start with, an extrema point is chosen as the 'start' point. Several extrema points closer to the 'start' point are chosen as 'candidates', and one of them is chosen as the 'destination' point. The vector of the arc between the start point and the destination point is calculated. Beginning with a cell one of whose vertex is the 'start' point, the arc is traversed. This is done until the cell containing the destination point is reached. Two classes, class Arc and class Graph are used to maintain information about the cells that are traversed after arc formation, and for holding the final extrema graph information. The cell ID of each traversed cell is inserted into a list. If the traversal crosses the boundary of the volume, the traversal is abandoned and the next available closest destination point is chosen for

starting a new traversal. Flags are maintained for every extrema point to check how it is connected to another extrema. To begin with, the flag of an extrema point is its grid point ID. When it is connected with another extrema, the flag that has the larger value is replaced with the flag ID of the extrema with a smaller flag ID value. In addition, flag values of other extrema that are connected to this one are substituted with the new small value. This is done so that when choosing a destination point an extrema point with the same flag value as the start point is not selected.

2.5.3 Generating Sorted Boundary Cell Lists

Boundary cells are defined as those cells in the volume that have one or more faces that are not connected to any other cell [8]. Two structures, a BCELL and a BLIST are used for maintaining sorted boundary cell list. The minimum and maximum values are defined for each boundary cell and two lists based on each cell's minimum and maximum are formed using a quick sort algorithm.

2.5.4 Generating Isosurfaces

Given a value q, the algorithm generates an isosurface corresponding to the value using the propagation algorithm. First, the seed set, i.e., the extrema graph and the sorted boundary cell lists, are searched for any cells that contain the specified isosurface. To begin with, the minimum value sorted cell list is traversed, only until the minimum value becomes greater than the value specified. If the maximum value of the visited cell is higher than the value specified, the cell is considered as an isosurface cell. The arcs are

searched next for any occurrence of isosurface cells. The given value is compared against the maximum-minimum value for each arc to check if the given value lies between these two values. If so, then all cells in that arc are visited. When an isosurface cell is found, its ID is put into a FIFO. After traversal of the cell list and the graph is completed, a cell is removed from the FIFO and the isosurface is generated for it. The cells that share a face with this cell are then pushed into this queue and the process is carried on recursively until the FIFO is empty (as described in the propagation algorithm in Section 2.4).

The number of cells in the boundary cell lists is estimated as $O(n^{2/3})$ [8], and the number of cells in the extrema graphs is estimated as $O(n^{1/3})$ [8]. The cost of isosurface generation for this algorithm is estimated as $O(n^{2/3})$ [8]. The number of cells intersected by the isosurface is estimated as $O(^{n1/3})$ [8]. The worst-case estimate for the number of arcs in this algorithm as given in [9] is $O(n)$. This happens when the data exhibits small perturbations and each node is an extrema, in which case each cell is an arc by itself.

2.6 CONTOUR TREE

The reviewed literature for the Contour Tree method [11] focuses on the procedure for finding isosurface cells using this method. It does not address the surface topology determination, vertex computation, or the surface normal computation steps in detail. Any polygonization algorithm could be used in conjunction with the Contour Tree method to complete the isosurface generation process for a given data set.

The Contour Tree method is also a seed set based method. The algorithm proposed by Kreveld [11], aims at connecting not only extremum points but also saddle points in an unstructured volume. In an unstructured volume, it is possible to have voids and through-holes. A through-hole is a topological feature that increases the genus[3] of an object. A void is an empty space enclosed by a disjoint part of the boundary of a volume [11]. Figure 2.2 shows a void and a through-hole in a volume.

Consider the case where a volume contains a through-hole. It is possible that an isosurface in the volume has 2 or more disjoint parts because of the existence of the through-hole. In this case, an Extrema Graph might not contain cells that intersect all the disjoint parts of the isosurface. In order to counter this flaw, the Extrema Graph has to either maintain a sorted boundary cell list or preserve the topology of the through-holes so that it intersects all the disjoint parts of every isosurface in the volume.

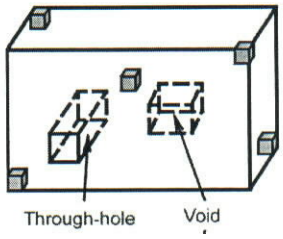


Figure 2.2: Through-hole and Void in a Volume (taken from [12])

[3] In common terminology, genus represents the number of holes in an object.

The contour tree method overcomes the need to maintain a sorted boundary cell list. It does this by connecting the saddle points in a volume using a tree in addition to connecting the extremum points. A saddle point is defined as a point that is a stationary point, but is not an extremum point [18]. The method generates a tree by traversing cells, starting from the local maximum points, merges or splits at the saddle points and terminates at the local minimum points. The cells in the contour tree are then traversed and cells are selected to form a seed set in a manner such that the scalar range across the tree is not missed out. The method requires over $O(n)$ [12] time for the computation of the tree structure. But the seed set formed is much less than the number of cells obtained from using extrema graphs and sorted boundary cell list. This successfully preserves all the topological features of the volume and a sorted cell list is not required.

2.7 EXTREMA SKELETON FOR FAST ISOSURFACE GENERATION

Itoh [10] describes in detail the steps involved in finding the isosurface cells using the Extrema Skeleton method. This method focuses on the acceleration of the cell selection stage and uses the Propagation algorithm to generate an isosurface.

This is a seed set based method that counters the problems associated with datasets containing voids and through-holes (discussed in Section 2.6) by using volume thinning (discussed in Section 2.7.1). The aim of the algorithm is to reduce the number of non-isosurface cells visited by forming a seed set. To begin with, the seed set is searched

to find isosurface cells. Using the isosurface cells thus found, the entire isosurface can be generated by isosurface propagation.

This is achieved by a two part preprocessing stage. In the first part, extremum points are extracted from the given volume using the method described in Section 2.5.1. The process involves visiting every cell and its adjacent cells once, to compare the scalar values and extract the extremum points. All cells that have any of these extremum points as one of their vertices are marked as extrema cells. Since the number of adjacent vertices of a vertex is constant for a given cell type, the computation time for extremum point extraction is $O(n)$ [12].

In the second stage, the extrema cells are connected by cells in the data set selected using the volume thinning method to form an extrema skeleton. This extrema skeleton intersects all the disjoint parts of any given isosurface in the volume. The extrema skeleton is formed only once for a given dataset. This skeleton can be used until the visualization application terminates.

The pseudo code [10] describing the various steps in the algorithm is shown in Figure 2.3. When an isovalue is specified by the user, the extrema skeleton is first searched to find the cells that are intersected by the isosurface. The neighbors of these cells are then visited recursively until the entire isosurface is extracted.

```
void main(){
      /* preprocessing */
      ExtractExtremumPoints();
      VolumeThinning();

      /* isosurfacing process */
      while(1){
            specify an isovalue;
            Extract_isosurfaceCells_from_skeleton();
            IsosurfacePropagation();
            }
}
```

Figure 2.3: Pseudo Code for the Extrema Skeleton Algorithm (taken from [10])

2.7.1 Summary of the Volume Thinning Method

The extrema skeleton algorithm is described for an unstructured dataset composed of hexahedral cells in general. The volume thinning method is an extension of the image thinning method [12]. Image thinning generates the skeleton of the pixels of a painted area while trying to preserve the features of the image itself.

The first stage marks all the extrema cells in the volume. The marked cells are retained as part of the extrema skeleton. To begin with, the implementation assumes that the skeleton is made up of all cells in the dataset. Each “unmarked” cell on the boundary of the volume is then visited and many of them are eliminated based on certain conditions discussed later in Chapter 3. In order to determine whether a cell needs to be retained as part of the skeleton, the nodes and edges shared by a cell’s neighbors are considered. If the shared node or edge is on the boundary of non-eliminated cells, then the cell under consideration is retained. If the node or the shared edge is on the inside, then it is possible to traverse from one neighbor cell to another through their shared nodes or edges in the absence of the cell being considered. The method finally generates a one cell wide skeleton for the given volume which contains extrema cells and cells that are necessary to preserve the topological features in a volume. The order in which the cells are visited determines the shape of the skeleton and the number of cells in it.

CHAPTER 3

IMPLEMENTATION FOR VTK

The Extrema Skeleton algorithm described in Section 2.7 was implemented for the graphics package Visualization Toolkit (VTK) [15], version 3.2.

3.1 ISOSURFACING IN VTK

3.1.1 Components of a Scene

The process of generating an image using a computer is called rendering [15]. For rendering images in 3D, there are techniques that simulate the interaction of objects (or actors) with lights and camera to generate images. A combination of actors, lights, and camera constitute a scene.

There are eight basic objects that are used to render a scene in VTK [15].

- *vtkRenderMaster*, creates a rendering window.
- *vtkRenderWindow*, manages a window on the display device. One or more renderers can draw into an instance of *vtkRenderWindow*.
- *vtkRenderer*, coordinates the rendering process involving lights, camera and actors.
- *vtkLight*, a source of light, illuminates the scene.
- *vtkCamera*, defines the view position, focal point and other viewing properties of the scene.

- *vtkActor*, represents an object rendered in the scene.
- *vtkProperty*, defines the appearance of an actor like color.
- *vtkMapper*, the geometric representation for an actor, interfaces the geometric structure to the graphics library.

Since visualization transforms data into images and accurately represents information about data, it deals with the issues of Transformation and Representation. Transformation is the process of converting data into graphics primitives for display, while representation deals with the internal representation of data and graphics primitives. Thus a visualization model is characterized by a functional model, and an object model. The functional model represents the transformations as functions. The object model specifies representations as objects. Examples of a functional and an object model are shown in Figure 3.1 and Figure 3.2 respectively.

In the functional model, the oval blocks indicate operations (processes) performed on data, and the rectangular blocks show the data stores (objects) that represent and provide access to data. Data stores are shown within two horizontal lines. Arrows leading into a block indicate input and arrows leading out of a block indicate output. The blocks sometimes have parameters or variables that serve as additional inputs. Processes that create data with no inputs are called Source Objects. Processes that only take data inputs and create no data outputs are called Sinks. Processes that consume data inputs and give as output transformed data are called Filters.

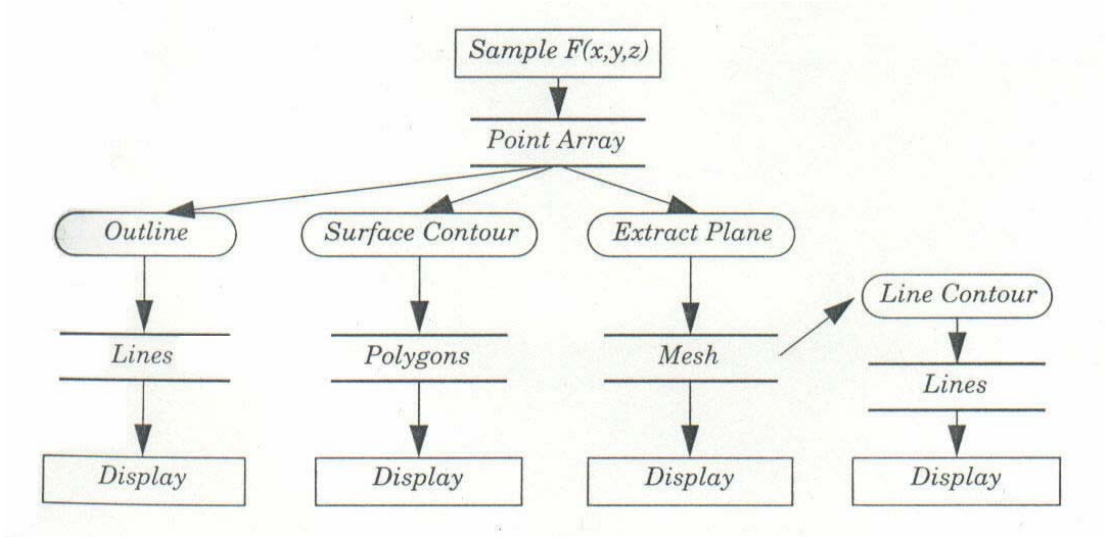


Figure 3.1: Functional Model (Image courtesy of Kitware, Inc. and taken from [15])

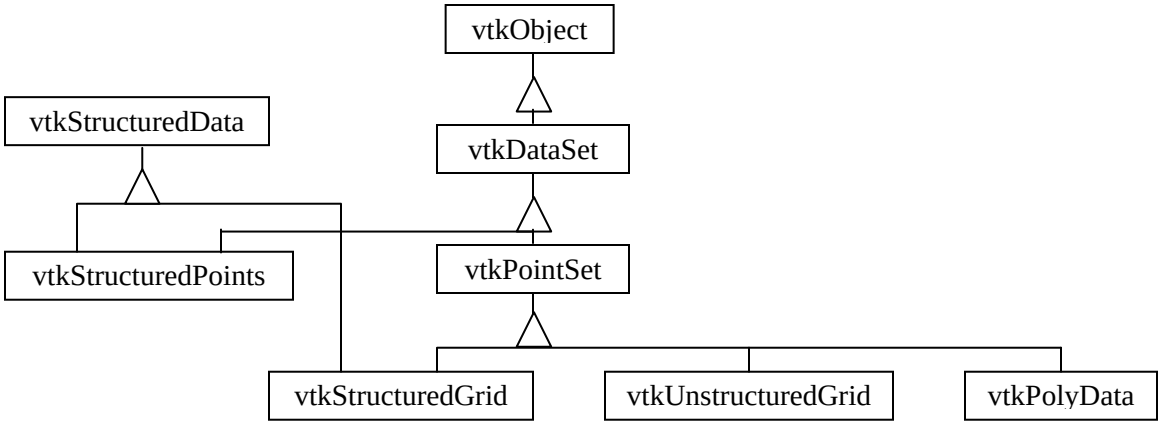


Figure 3.2: Example of Object Model (Image courtesy of Kitware, Inc. and taken from [15])

The object model represents the objects in the system, their properties and their relationship with other models in the system. In VTK, the Object Modeling Technique (OMT) developed at GE is used [15]. In the OMT representation, object classes are represented as solid rectangles and instances of objects are represented as dotted rectangles. The object name, variables and methods that operate on the object are given inside the rectangle. Each rectangle has lines dividing it into sections that show the object name, variables and methods separately. Line segments are drawn between related objects to represent the relationship between them. Inheritance is depicted by a triangle, with the parent class at the apex of the triangle and the derived class at the base of the triangle. The inheritance tree grows top-down. Figure 3.2 shows an example of the dataset object model in VTK.

3.1.2. Visualization Pipeline

The functional model of data visualization is also referred to as the visualization pipeline or the visualization network. The pipeline consists of objects to represent data (data objects) and objects to operate on data (process objects). Figure 3.3 shows a visualization pipeline.

Data objects represent information. They provide methods to create, access and delete information. Direct modification of information is not allowed except through formal object methods.

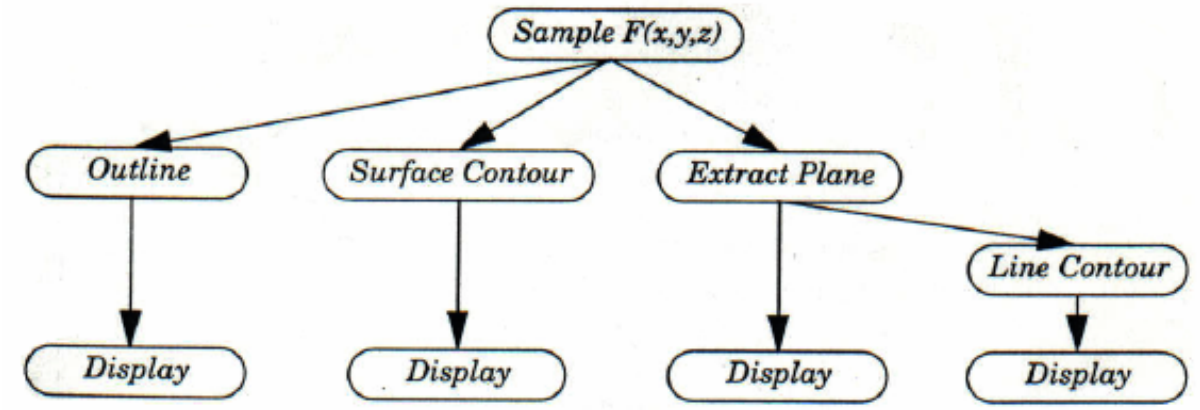


Figure 3.3: Visualization Pipeline (Image courtesy of Kitware, Inc. and taken from [15])

Process objects are objects that operate on data inputs and generate data outputs. They are further classified into Source objects, Filter objects or Mapper objects, depending on whether they initiate, maintain or terminate the data flow in the visualization pipeline. Source objects that interface to data stored in files are called Reader objects. They read data from an external file and produce a data object. Filter objects require one or more data inputs and generate one or more data outputs. Mapper objects correspond to sinks in the functional model. They consume one or more data inputs and terminate the data flow in the pipeline. Mapper objects usually transform data inputs into graphics primitives. They can also write data to a file or interface with another software system.

3.1.3 Executing the Pipeline

The complete process of causing each process object to operate is called the execution of the network or the pipeline [15]. In general, a process object should be executed only when all its input objects are up-to-date. This requires synchronization of all the process objects in the network, beginning with the source object. This synchronization is done by explicit control or by implicit control as shown in Figure 3.4.

Explicit control means directly controlling the execution of the process objects. A centralized executive is used to track the changes to the network, make explicit dependency checks to coordinate the network, and execute only those objects whose inputs have changed .

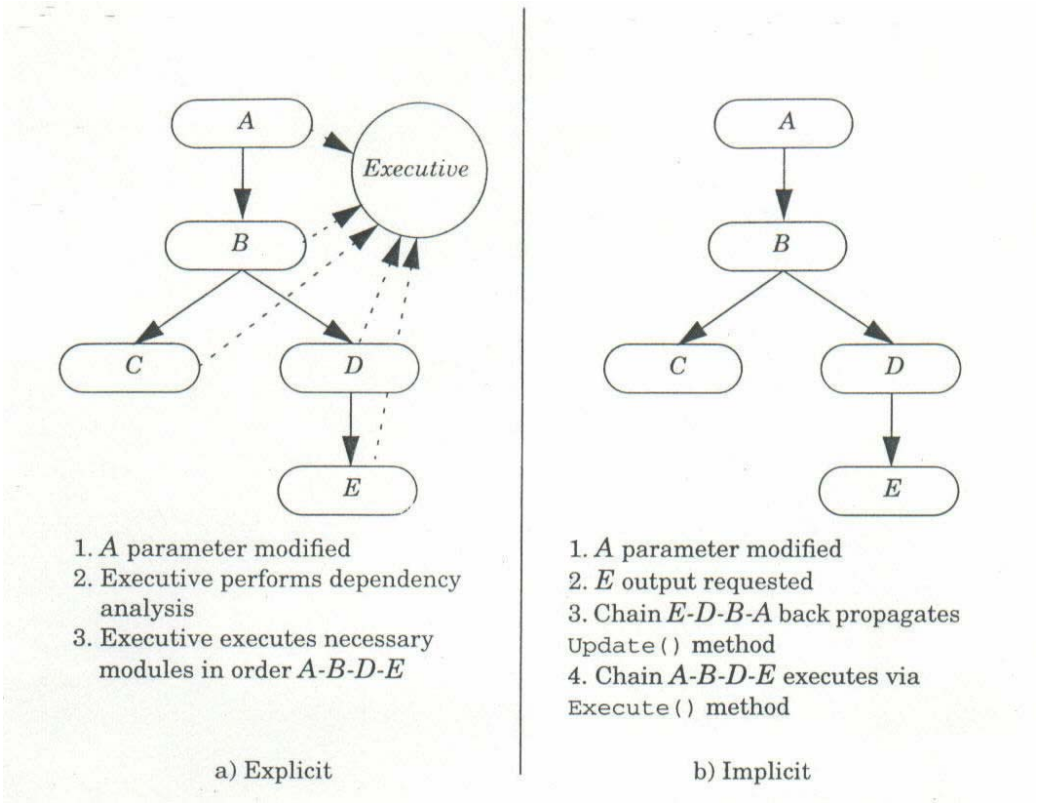


Figure 3.4: Explicit and Implicit Network Execution (Image courtesy of Kitware, Inc. and taken from [15])

Implicit execution means that an object is executed only if its local input parameters change. The two main steps involved in implicit execution are Update() and Execute(). Referring to figure 3.4(b), when output is requested from object E, it requests input from its input object D. Chain E-D-B-A back propagates the Update() method until the source object A is encountered. The source object A then executes if there is a change in itself or its external inputs. The output from A is then passed to its requesting object B. B calls the Execute() method since its input has changed. Thus, every object in the chain A-B-D-E checks to see whether itself or its inputs have changed, and executes via the Execute() method. The chain ends when the initial requesting object E executes and terminates.

Scalar isosurfacing in VTK is done using implicit execution. The filter object that is used to extract isosurfaces in VTK is the *vtkContourFilter* class. *vtkContourFilter* class is a general filter used for generating isosurfaces. This class object accepts any dataset as input and generates polygonal data, like triangles, as output. While the *vtkContourFilter* class can deal with any cell type, it is up to the individual cell type class to provide the method for contouring (isosurfacing in this case). The pseudo code for generating an isosurface is shown in Figure 3.5. The isosurface generation process begins when the actor sends a request to be rendered.

```
int main(int argc, char** argv) {
    create_render window_to_display_resulting_image() ;
    create_renderer() ;
    read_datafile_using vtkStructuredGridReader() ;
    /** output of above is structured grid **/
    create_vtkContourFilter object();
    set the structured grid as input to the object();
    create a mapper();
    create actor();
    set mapper as actor's mapper();
    set the output of the vtkContourFilter object as the
    input of mapper();
    add actor to renderer();
    render the scene();
    }
```

Figure 3.5: Pseudo Code for Generating an Isosurface in VTK

3.2 IMPLEMENTATION OF THE EXTREMA SKELETON ALGORITHM FOR VTK: DETAILS AND METHODOLOGY

The extrema skeleton algorithm was adapted and implemented for simple structured grids composed of voxel cells in VTK. The volume had no voids or through-holes. The strategy was to keep the flow of execution the same as in VTK and deviate only at the point of isosurface generation. Therefore, the implementation started with using the same structured grid reader, render window, renderer, light, mapper and actor classes as in the regular implementation using VTK. The *vtkContourFilter* class however was modified slightly for use with the Extrema Skeleton Method. For any given isovalue, the seed cell list was first searched for the occurrence of isosurface cells. Isosurface propagation was then used to find all remaining isosurface cells. These cells were passed to the *vtkContourFilter* class for polygonization.

Figure 3.6 summarizes the flow of data and the logical steps of the implementation in both VTK and in the Extrema Skeleton method. The flow of the program can be summarized using the pseudo code shown in Figure 3.7. The algorithm was implemented in C++. VTK has many interfaces for programming, like tcl/tk and Java. The possibility of programming in Java was explored, since Java has the advantages of garbage collection. However, there are limitations with regard to deriving classes, and accessing class variables. The number of functions available to program through the interface is also somewhat limited.

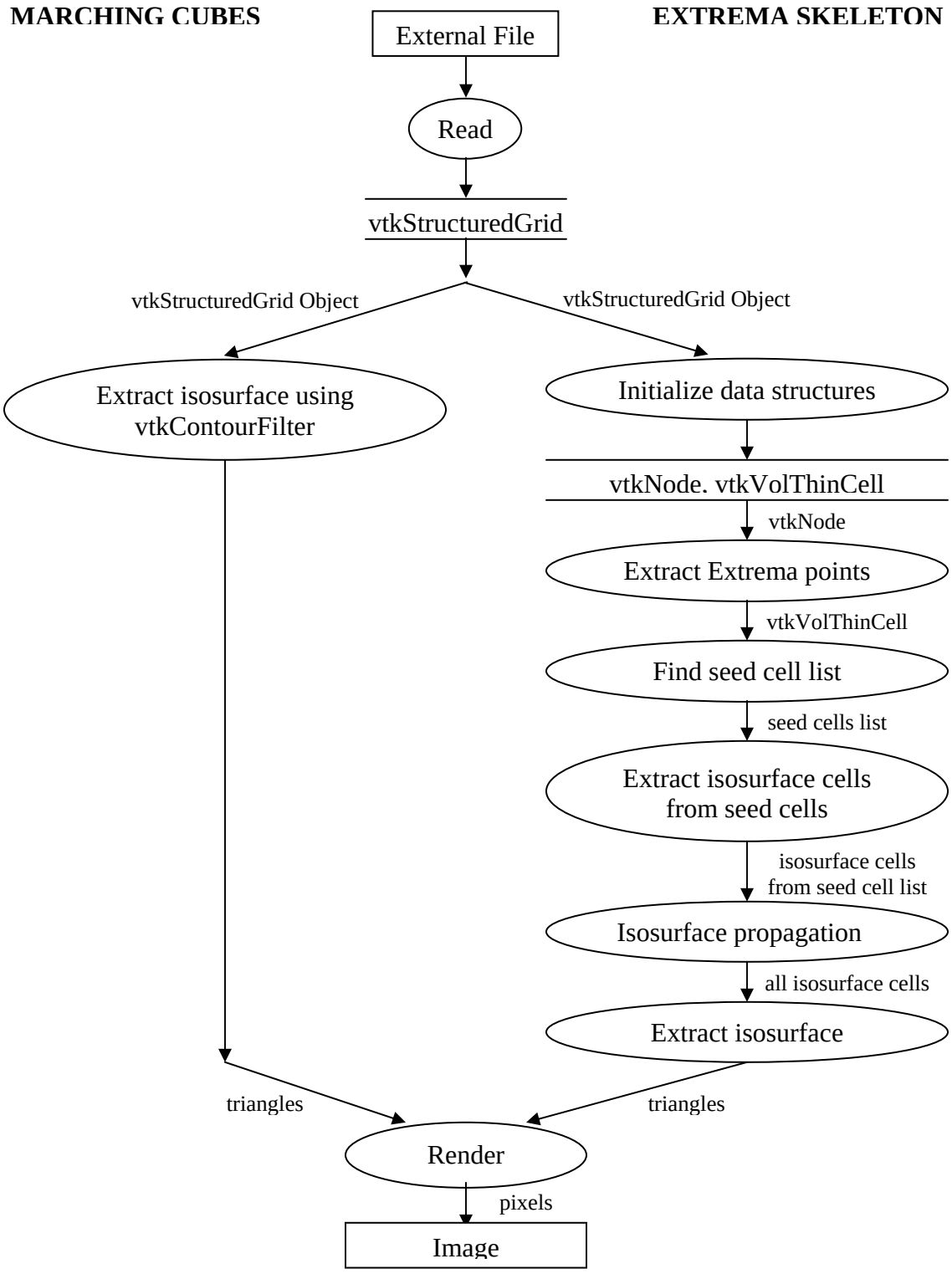


Figure 3.6: Implementation of Marching Cubes and Extrema Skeleton Methods in VTK

```
void main(int argc, char** argv) {
    create render window to display resulting image ;
    create renderer ;
    read datafile using vtkStructuredGridReader ;
    /** preprocessing for forming extrema skeleton **/
    create and initialize data structure to extract extremumPoints ;
    create and initialize data structures to form seed cell list ;
    extract Extremum points ;
    form Seed Cell List ;
    /** end preprocessing **/
    create contour filter ;
    setinput of contour filter as output of vtkStructuredGridReader ;
    for(i=0; i < number of contours; i++) {
       find cells corresponding to value i from Seed Cell List;
       queue neighbors of above found cells; //isosurface propagation
       }
     pass isosurface cells found above to modified vtkContourFilter ;
     create mapper ;
     set output of vtkContourFilter as input to mapper ;
     create actor ;
     set above mapper as actor's mapper ;
     add actor to renderer() ;
     render scene ;
    }
```

Figure 3.7: Pseudo Code for Implementation of Extrema Skeleton Algorithm in VTK

3.2.1. Data Structures Used for the Extrema Skeleton Method

The *vtkContourFilter* class was modified and a private member variable *IsIsoCell* of type *vtkIdList* was added. This list was used to store the IDs of all isosurface cells. The *vtkContourFilter::Execute()* function was modified to use the *IsIsoCell* ID list and polygonize isosurface cells only. The following classes were also used in the implementation for VTK:

vtkNode

This class represents a point on the grid and is used mainly to record the extremum points in the data set. It has scalar information pertaining to a grid point and has the following member variables:

- A float variable called '*scalar*' which holds the scalar value of each grid point.
- Boolean variables *Max, Min*. - A grid point is an extremum point if it has only one of its boolean variables set to true. The position of the node is stored in the internal representation of a cell in VTK and so this information is not duplicated here.

vtkVolThinCell

This class represents the type of cells the volume is made of. A cell e_i has the following variables associated with it :

- *neighborList* - an ID list of type *vtkIdList*, which contains the index of each of the adjacent cells, $e_{i,1}$, $e_{i,2}$, …, $e_{i,6}$ of cell e_i. Each non-existent neighbor is represented by –1 in the list.
- *'eliminated'* – a boolean variable to indicate whether or not a cell has been eliminated from the extrema skeleton.
- *CellType* – an integer that shows the number of neighbors the cell has.

vtkExtremumPtsList

The functions of this class are used to find the extremum points in the data set and has the following member variable :

- *extremumPtsList*, of type *vtkIdList*, used to store the ID of the extremum points in the data set.

vtkSeedCellList

The functions of this class are used to form the extrema skeleton. The member variables are integer arrays, of type *vtkIdList*, that are used to store or point to ID of cells and vertices. They are listed below.

- *seedCellList*, of type *vtkIdList*, used to store the ID of the cells selected to form the seed list.
- *neighbList*, used to point to the neighbor IDs of a cell.
- *ptCellList*, used to point to the ID of all the cells that have a particular grid point as one of their vertices (i.e., share a vertex).

- *nodeList*, used to point to the ID of all the nodes of a cell.

neighbList, ptCellList and *nodeList* are all pointers of type *vtkIdList*. No memory is allocated for these members.

3.2.2 Implementation Procedure

In the main loop of the program, both the *vtkNode* and the *vtkVolThinCell* structures were initialized. Using the functions in the *vtkStructuredGrid* class, the scalar value for each point on the grid was retrieved from the *vtkScalars* data structure. This was used to initialize the *vtkNode* data structure. The neighbors for each cell in the volume were determined by finding out the list of cells that share all four nodes on the face of a cell. This information was stored in the *neighbList* member variable of the *vtkVolThinCell* class instance of that cell. A cell was classified based on the number of neighbors it had and also whether the cell was an extrema cell [10]. Accordingly, cells with no neighbors were of type *'0'*, cells with one neighbor were of type *'1'*, cells with two neighbors were of type *'2'*, and so on. All extrema cells were classified as *'-1'*. So, a voxel cell could be of type *'i'= -1,0, 1, 2,..., 6* . The cell type *'i'* for each cell was also stored in the *vtkVolThinCell* class instance of the respective cell.

Finding extremum points

Extremum points were extracted using the method described in Section 2.5.1.

Forming the extrema skeleton

To start with, all cells touching the extrema points were marked as type *'-1'*. These cells were retained as part of the skeleton and were not removed until the skeleton was formed. In the initialization stage, the boolean variable '*eliminated*' in the *vtkVolThinCell* class instance of all cells was set to false and all cells were assumed to be a part of the extrema skeleton. Each cell was then visited in the order in which it occurred in the structured grid object in VTK and was considered for elimination. If the cell was of type *'-1'* then its ID was added to the seedCellList array. All other cells were considered for elimination based on the conditions shown in Figure 3.8 [10]. The connectivity between a cell and its non-eliminated neighbors was checked. If a cell's non-eliminated neighbors could be traversed through their shared faces without having to pass through the cell in question, the cell in question was eliminated. For example, if e_i is the cell that had non-eliminated neighbors e_j and e_k, and if e_k could be reached from e_j by traversing through their shared faces without traversing through e_i, then e_i can be eliminated. When a cell was eliminated, its cell type was changed to *'0'*, its boolean variable *'eliminated'* was set to true, and the cell type of all its non-eliminated neighbors was changed from *'i'* to *'i-1', i>0*. All type '*0'* and type *'1'* cells were automatically eliminated since there were no conditions to be checked for and since removing them did not affect the connectivity of their neighboring cells. Since the cell type of neighbor cells was continually updated during the thinning process, conditions for type *'6'* cells were not required, because the type *'6'* cells would change to some other cell type during the process.

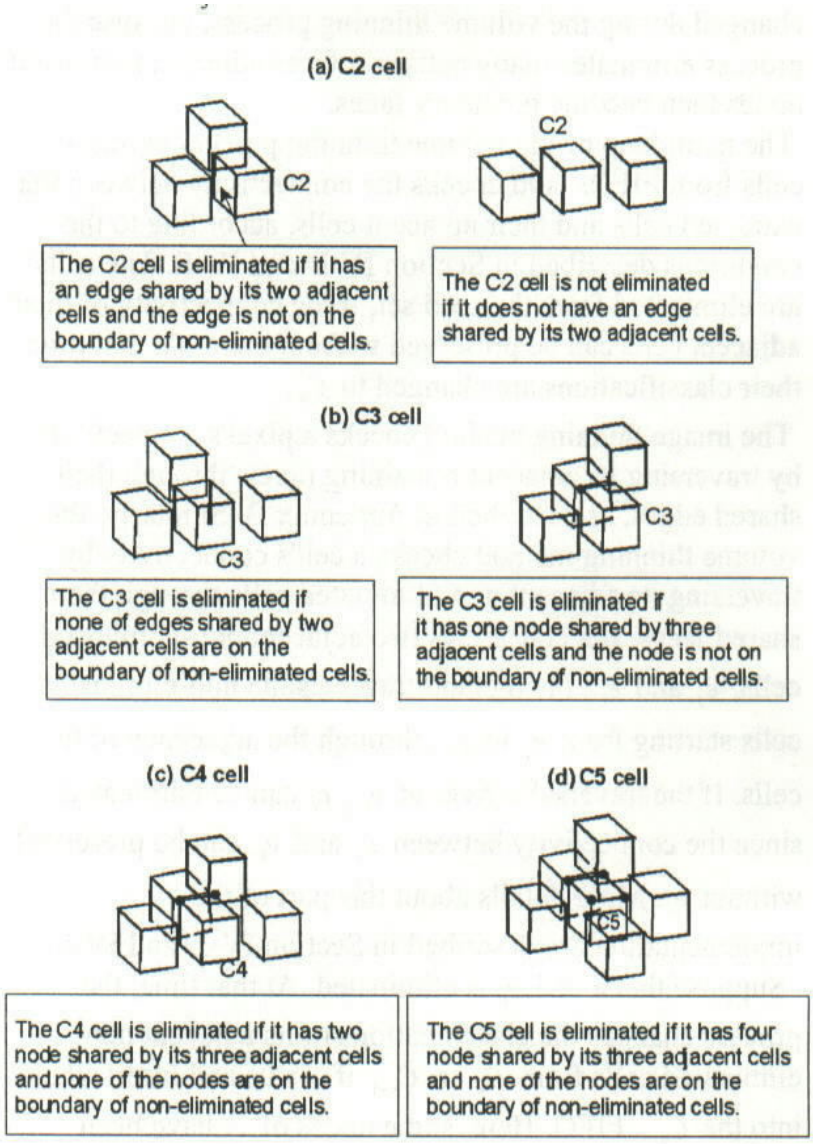


Figure 3.8: Conditions for Hexahedral Cells (taken from [10])

Finding isosurface cells from the skeleton

Given a value, the seed cell list was searched for any cells intersected by the surface of interest. Upon finding 'hit' cells, the IDs of all such cells were stored in an integer array of type *vtkIdList*. These cells were then used as the starting points for isosurface propagation, described in Section 2.4, to find all the isosurface cells. This final list of isosurface cells was set as the *IsIsoCell* list member variable of the *vtkContourFilter*.

3.3 POLYGONIZATION

The *vtkContourFilter* class is used to generate surface contouring primitives (points, lines, polygons) depending on the input cell type. In the case of voxel cells, the primitives are triangles. The usual path for executing polygonization in VTK can be summarized as follows:

The two stages – cell selection stage, and polygonization stage are executed in the same function. Upon finding an isosurface cell, the surface inside the cell is determined for all desired isovalues. The *vtkContourFilter* class initializes all the necessary data structures for creating and storing polygonal data. Then, for each cell in the data set and for each isovalue, it calls the *Contour()* function of the appropriate cell type (in this case, *vtkHexahedron::Contour()*). The contour method for the *vtkHexahedron* class creates an index based on which vertices of the cell are outside or inside the surface (i.e., based on whether the scalar at the vertices of the cube is greater than, less than, or equal to the desired isovalue). This index is then used as a pointer to look up the appropriate list of

intersected edges for the respective case in a table of surface-edge intersections. Using this entry the point of intersection of the surface along the respective edges is found using linear interpolation. The new points so created are checked for uniqueness using *vtkMergePoints* object.

The *vtkMergePoints* class is a concrete implementation of the abstract class *vtkLocator*. The abstract class, *vtkLocator,* is a spatial search object used to locate points in 3D. It works by dividing a specific region in space into a regular array of "rectangular buckets". Each bucket contains a list of points. These buckets can be quickly found in response to queries like point location. Locator objects are generally organized as tree structures and typically work as follows. Points or cells are first inserted into the tree structure. A point or cell will be associated with a certain bucket. When geometric operations are performed, they are first performed on the buckets, and if the operation turned out positive, more expensive operations are performed on the points and cells in the bucket.

After the newly created points are checked for uniqueness using the *vtkMergePoints::InsertUniquePoint()* function, sets of points that form non-degenerate triangles (triangles whose vertices are non-coincident) are inserted into a *vtkPolyData* object. The *vtkPolyData* class is a concrete implementation of the *vtkDataset* class. It represents a geometric structure consisting of vertices, lines, polygons and triangle strips.

Point attributes, such as scalars, are also represented [15]. The *vtkPolyData* object holds the newly created polygonal data, i.e., vertices of the newly created triangles.

3.4 RESOLVING PERFORMANCE PROBLEMS

In the implementation of the Extrema Skeleton algorithm for VTK, the same procedure for generating polygonal data described in Section 3.3 was used. However, instead of applying the procedure on all the cells in the dataset (which is the case in Marching Cubes), only the isosurface cells that were chosen using the Extrema Skeleton algorithm underwent polygonization. When the list of isosurface cells was passed as input to *vtkContourFilter*, the polygonization time taken was longer than expected. This delay was attributed to the *vtkMergePoints::InsertUniquePoint* function. The *InsertUniquePoint* function checks for uniqueness of newly created points of intersection of the surface on cell edges. The delay was found to be due to insufficient memory allocation (memory allocation was proportional to the number of isosurface cells) to the input parameters to the *vtkMergePoints* object. To overcome this problem, the input parameters were initialized with memory proportional to the total number of cells in the dataset. This change resulted in a significant reduction in the polygonization time.

CHAPTER 4

RESULTS AND DISCUSSION

The implementation of the extrema skeleton method for VTK was compiled and run on the Redhat Linux 7.2 operating system on a PC with a 500 MHz Pentium III processor and 512 MB of RAM. The VTK implementation of isosurface generation using the *vtkContourFilter* class was compared against the implementation of the Extrema Skeleton method for VTK in terms of the time taken to generate multiple isosurfaces, the number of cells searched for finding the isosurface and the time taken for the preprocessing stage. The timing scheme for both methods is shown in Figure 4.1. The *getrusage* system call was used to obtain total time for contouring and rendering the resulting image. g*etrusage* returns the current resource usage for the program that calls it. The syntax for this function is *getrusage (int who, struct rusage *usage)* where *who* is either *RUSAGE_SELF* or *RUSAGE_CHILDRE*N. This function contains several member variables that provide useful information on system resource usage for the calling program. One such member variable *timeval ru_utime* was used to record the user time taken until *getrusage* was called in the program. Time taken to execute a set of statements in the program was determined by taking the difference between the recorded time value before the first statement and the recorded time value after the last statement.

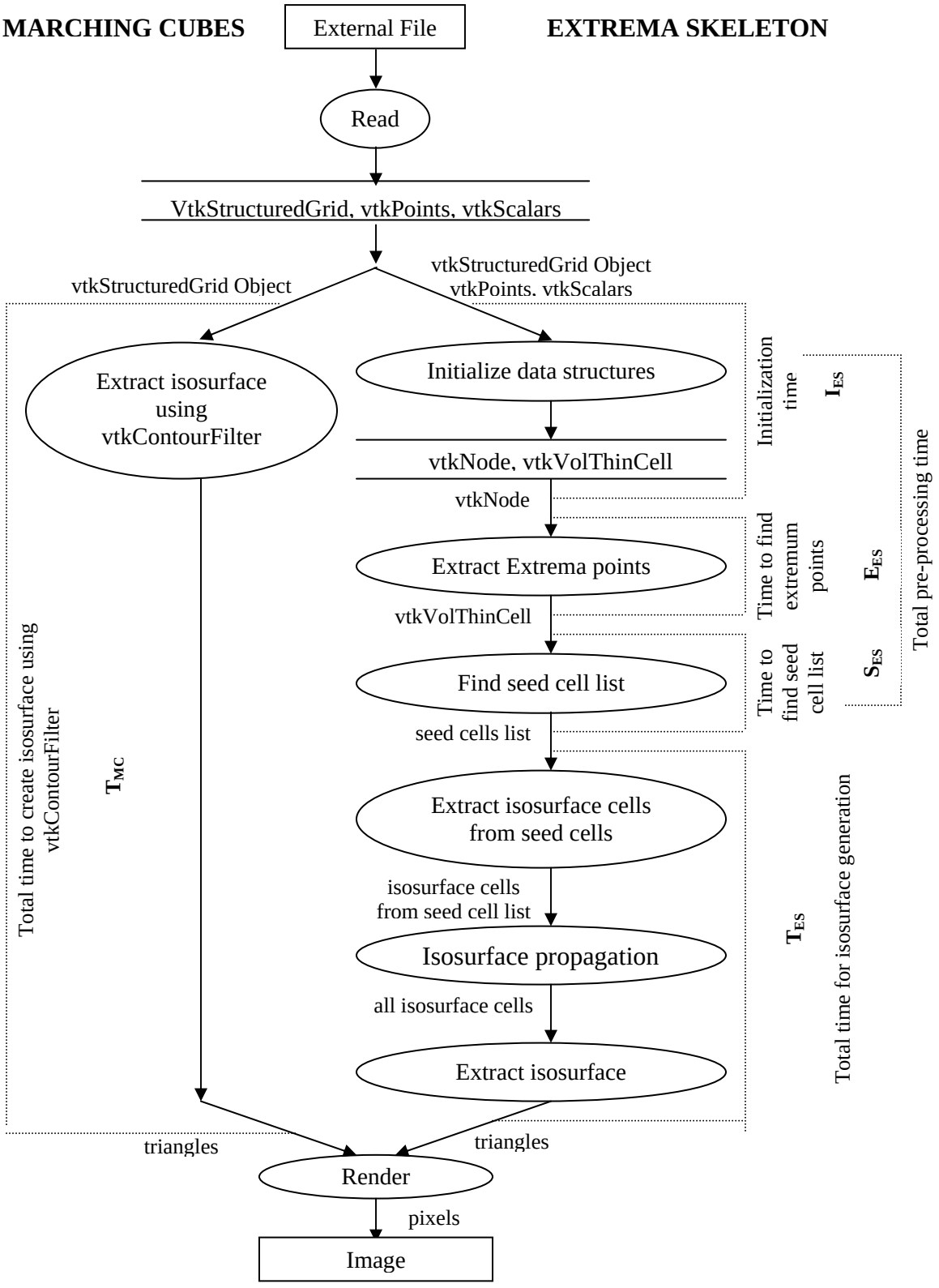


Figure 4.1: Timing Scheme for Comparing the Marching Cubes Algorithm with the Extrema Skeleton Algorithm

4.1 DATASETS TESTED

Table 4.1 gives the sizes of the datasets used for comparing the two methods. The datasets Ellipsoid.vtk, Hyperboloid.vtk, and Paraboloid.vtk were generated by taking structured grids of differing sizes and assigning scalar values to each point in the grid using equations for an ellipsoid, hyperboloid, and paraboloid respectively. Equations and ranges used to generate these datasets are shown in Appendix A. FlowFrame1.vtk and FlowFrame20.vtk contain real data that were provided by the Computational Mechanics Research Group (CMRG) at the University of Tennessee Space Institute, Tullahoma. These datasets were generated by CMRG from studying the flow of water around a solid object. AllIso.vtk was generated to compare the two algorithms in the case where all cells in the dataset were isosurface cells. The scalar value for every point on the grid of this dataset was either –1 or 1, and no two consecutive points on the grid had the same value. Thus, if an attempt were made to generate isosurfaces between values –1 and 1, all cells would be isosurfaces cells. The procedure used to generate AllIso.vtk was also used to generate ZeroIso.vtk and SomeIso.vtk. However, when testing ZeroIso.vtk the isovalues to be searched were chosen such that none of the cells in the dataset contained the search values. In the dataset SomeIso.vtk, less than 1% of the scalar values were arbitrarily changed so that some of the cells in the dataset contained the search values.

Figure 4.2 shows some of the properties of the datasets tested. The graph shows the fraction of seed cells and the fraction of isosurface cells for each dataset. Small seed cells fraction indicates that the dataset is simple with a regular well-defined distribution

Table 4.1: Size of Datasets

Dataset #	No. of Points	Number of Cells
ZeroIso.vtk	293,301	280,000
SomeIso.vtk	293,301	280,000
Ellipsoid.vtk	23,001	20,480
Hyperboloid.vtk	58,466	54,000
Paraboloid.vtk	69,741	64,000
FlowFrame1.vtk	371,385	353,600
FlowFrame20.vtk	371,385	353,600
AllIso.vtk	52,521	48,000

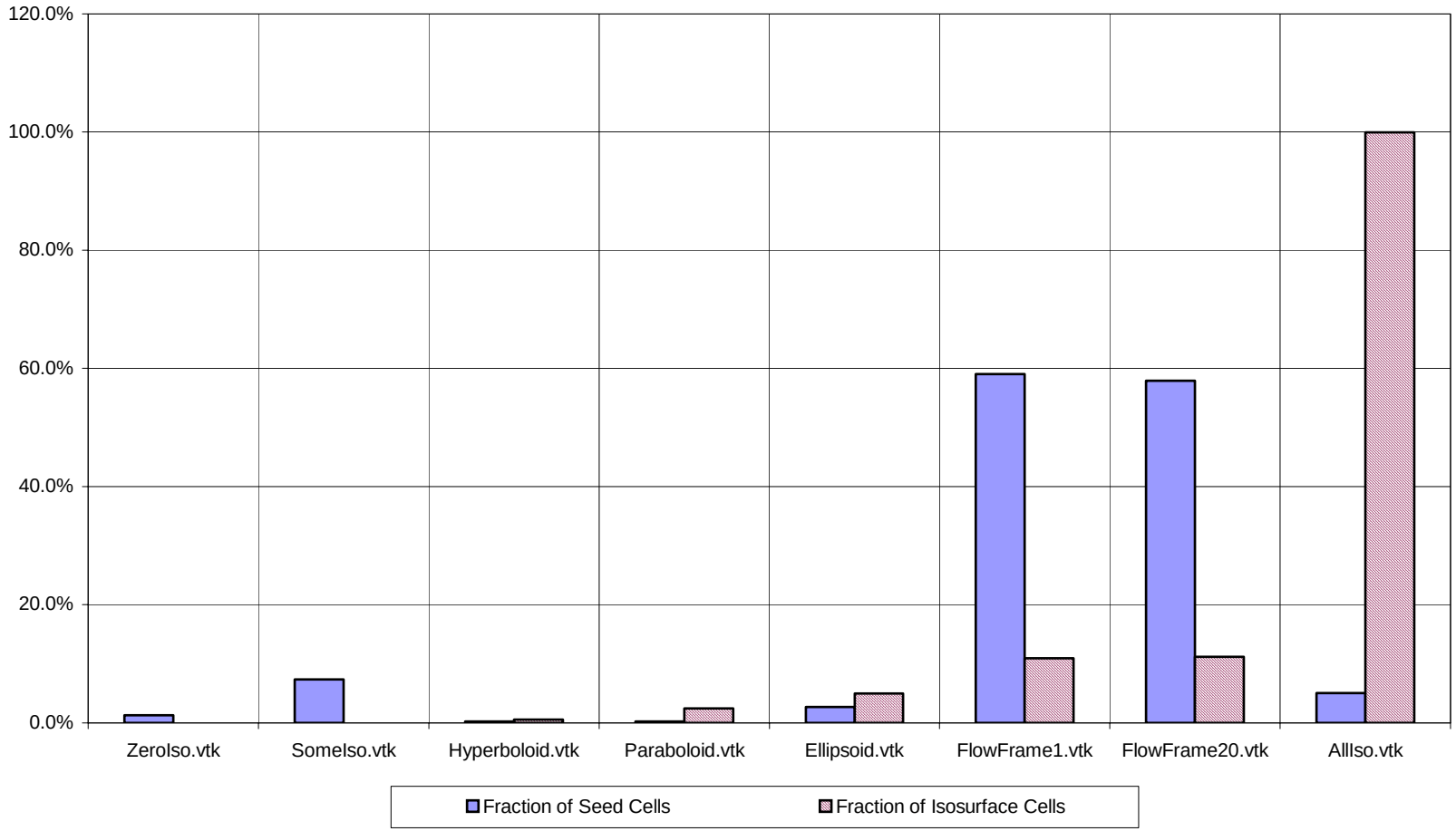


Figure 4.2: Dataset Characteristics

of scalar values in the grid. Large seed cells fraction indicates that the dataset is complex with a significant fluctuation of scalar values from cell to cell. The fraction of isosurface cells in a dataset depends on the scalar values and the isosurfaces being extracted. The datasets ZeroIso.vtk, SomeIso.vtk, Hyperboloid.vtk, Paraboloid.vtk, and Ellipsoid.vtk are simple datasets[4] with less than 10% isosurface cells. The datasets FlowFrame1.vtk and FlowFrame.vtk are complex datasets[5] with more than 10% isosurface cells. The dataset AllIso.vtk is a simple dataset with 100% isosurface cells.

Isosurfaces extracted from each dataset tested except ZeroIso.vtk and SomeIso.vtk are shown in Appendix B. No polygons were generated for ZeroIso.vtk because no isosurface cells were found in the grid. Very few polygons (corresponding to disjoint parts of an isosurface) were generated for SomeIso.vtk to be visualized. Additional datasets based on regular shapes like sphere and cylinder were also generated and visualized by extracting multiple isosurfaces. These are shown in Appendix C.

4.2 RESULTS

The time taken for the various preprocessing stages is given in Table 4.2. The number of extrema points and the number of seed cells for each dataset is given in Table 4.3. Ten isosurfaces were generated for each dataset. Table 4.4 shows the number of isosurface cells present in each dataset. The table also compares the total number of cells searched by each method. Table 4.5 lists the time taken to find all the isosurfaces plus

[4] Datasets with <50% seed cells.
[5] Datasets with >50% seed cells.

Table 4.2: Time for Various Preprocessing Stages (microseconds per cell)

Dataset #	I_{ES}	E_{ES}	S_{ES}	$TPP_{ES}=I_{ES}+E_{ES}+S_{ES}$	TPP_{MC}
ZeroIso.vtk	11.64	6.50	6.04	24.18	0
SomeIso.vtk	11.64	6.46	6.39	24.50	0
Ellipsoid.vtk	12.04	6.30	5.74	24.07	0
Hyperboloid.vtk	11.88	6.25	5.94	24.06	0
Paraboloid.vtk	11.72	5.86	5.86	23.44	0
FlowFrame1.vtk	11.79	6.84	6.48	25.11	0
FlowFrame20.vtk	11.65	6.87	6.48	25.00	0
AllIso.vtk	12.29	6.46	6.04	24.79	0

I_{ES} – Initializing data structures for extrema skeleton method
E_{ES} – Time for finding extremum points for extrema skeleton method
S_{ES} – Time for finding seed cell list for extrema skeleton method
TPP_{ES} – Total preprocessing time for extrema skeleton method
TPP_{MC} – Total preprocessing time for marching cubes method

Table 4.3: Seed List and Extremum Points for the Extrema Skeleton Method

Dataset #	No. of Extremum Points	No. of Cells in Seed List
ZeroIso.vtk	51	3,578
SomeIso.vtk	64	20,591
Ellipsoid.vtk	2	551
Hyperboloid.vtk	2	113
Paraboloid.vtk	2	138
FlowFrame1.vtk	11,460	208,806
FlowFrame20.vtk	11,239	204,873
AllIso.vtk	41	2,418

Table 4.4: Number of Isosurface Cells and Cells Searched for Ten Isosurfaces

Dataset #	No. of Isosurface Cells	No. of Cells Searched	
		Extrema Skeleton	Marching Cubes
ZeroIso.vtk	0	35,780	2,800,000
SomeIso.vtk	92	206,680	2,800,000
Ellipsoid.vtk	10,302	99,950	204,800
Hyperboloid.vtk	3,288	43,000	540,000
Paraboloid.vtk	15,490	81,190	640,000
FlowFrame1.vtk	388,548	2,816,010	3,536,000
FlowFrame20.vtk	397,619	2,805,020	3,536,000
AllIso.vtk	480,000	480,000	480,000

Table 4.5: Time for Generating Ten Isosurfaces (microseconds per cell)

Dataset #	Extrema Skeleton	Marching Cubes
	T_{ES}	T_{MC}
ZeroIso.vtk	0.96	16.14
SomeIso.vtk	1.93	16.18
Ellipsoid.vtk	2.96	17.04
Hyperboloid.vtk	6.56	19.69
Paraboloid.vtk	16.11	23.44
FlowFrame1.vtk	38.07	40.16
FlowFrame20.vtk	38.72	40.72
AllIso.vtk	214.17	212.50

T_{ES} – Time taken by extrema skeleton method for generating isosurfaces.
T_{MC} – Time taken by marching cubes method for generating isosurfaces.

the time taken for polygonization for both methods. All times are expressed in units of microseconds per cell in the dataset. The number of polygons created for each dataset by both algorithms is shown in Table 4.6. Graphs based on the tabulated results are shown in Figures 4.3 through 4.5. Graphical and tabulated results are discussed in Section 4.3.

4.3 DISCUSSION OF RESULTS

Based on Figure 4.3, it is evident that the total preprocessing time per cell for the Extrema Skeleton method is constant. Therefore, preprocessing time is proportional to the number of cells in the dataset. For a given dataset, the preprocessing time is the same irrespective of how many isovalues are searched.

Table 4.6: Number of Polygons Generated for Ten Isosurfaces

Dataset #	Extrema Skeleton	Marching Cubes
ZeroIso.vtk	0	0
SomeIso.vtk	92	92
Ellipsoid.vtk	20,600	20,600
Hyperboloid.vtk	6,571	6,571
Paraboloid.vtk	30,229	30,229
FlowFrame1.vtk	890,358	890,358
FlowFrame20.vtk	914,258	914,258
AllIso.vtk	960,000	960,000

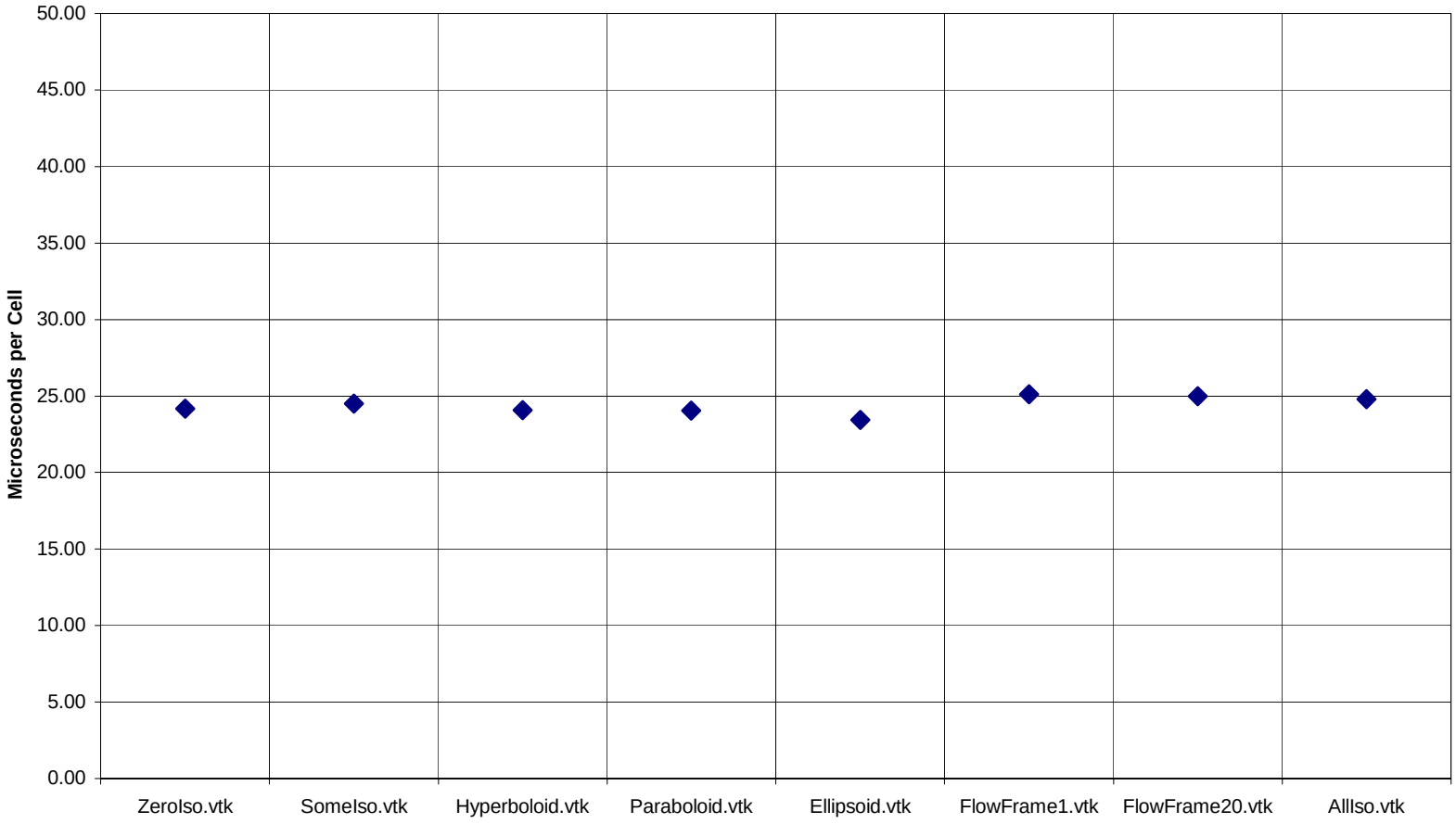


Figure 4.3: Preprocessing Time for the Extrema Skeleton Method

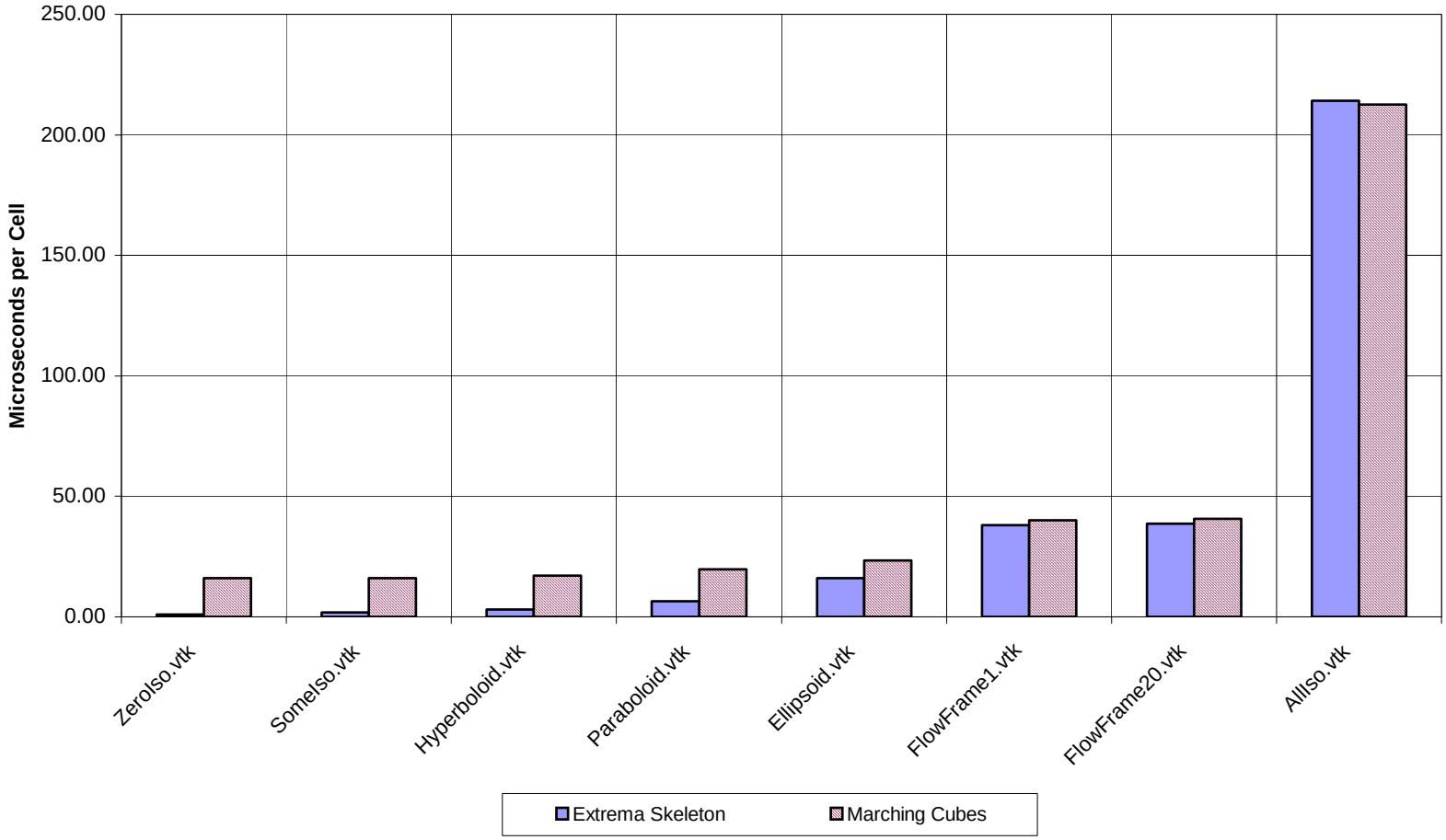


Figure 4.4: Comparison of Time for Generating Ten Isosurfaces

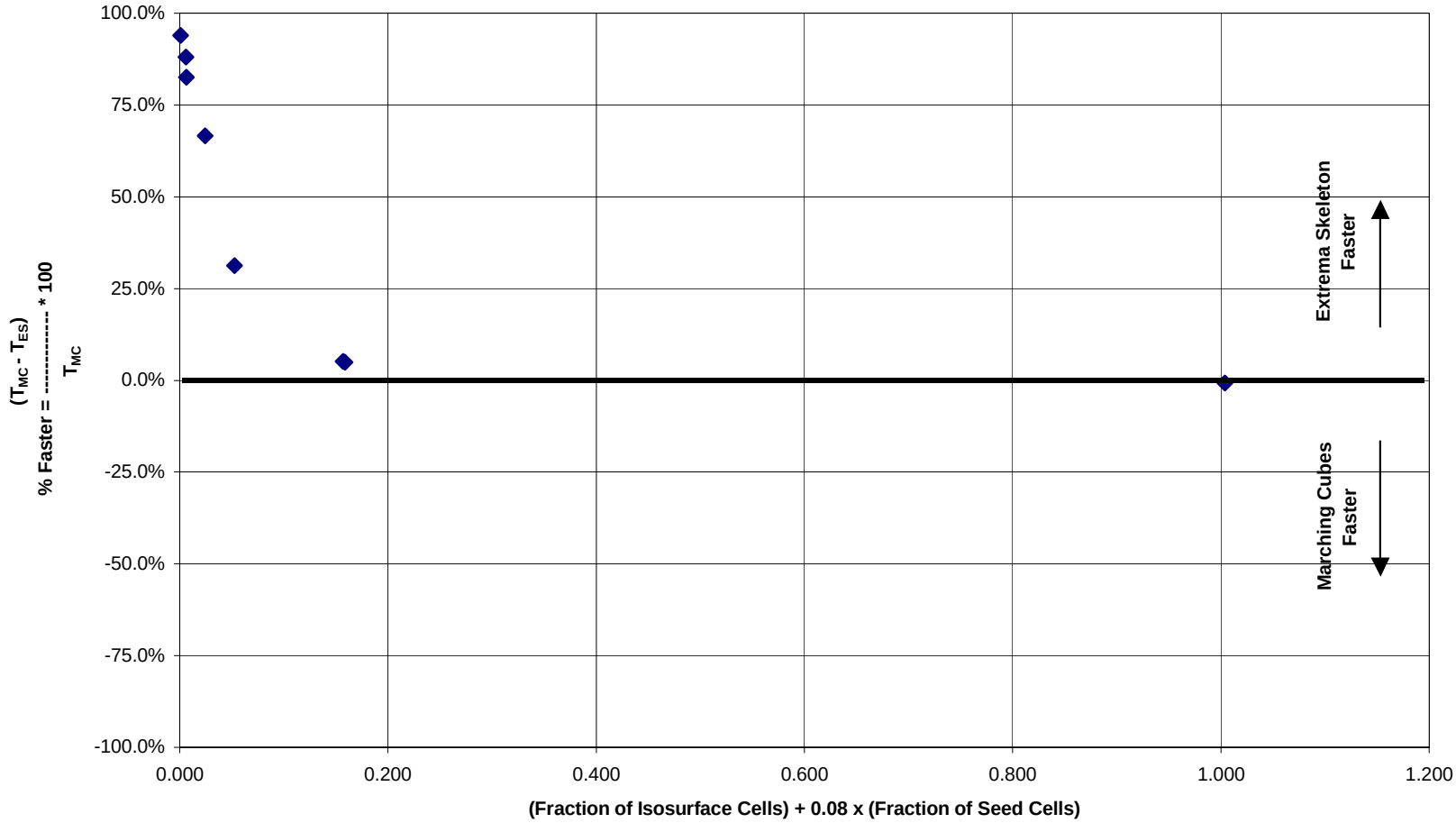


Figure 4.5: Performance of Extrema Skeleton Method

For all datasets except AllIso.vtk, the number of cells searched by the Extrema Skeleton algorithm for finding all isosurface cells was found to be less than the number of cells searched by the Marching Cubes algorithm. Since all cells in AllIso.vtk were isosurface cells, the total number of cells searched by both algorithms was the same. The results are shown in Table 4.4.

Table 4.5 compares the time taken by each algorithm to generate ten isosurfaces on different datasets. The results are plotted in Figure 4.4. This figure indicates that while the Extrema Skeleton method was significantly faster for some datasets, the time taken was comparable but still less than the Marching Cubes method for other datasets. Performance of the Extrema Skeleton method depends on two parameters - the fraction of seed cells and the fraction of isosurface cells in the dataset. The time taken by the Extrema Skeleton method is in fact the sum of the time taken to find isosurface cells (corresponds to the fraction of seed cells plus fraction of isosurface cells) and the time taken to polygonize the isosurface cells (corresponds to the fraction of isosurface cells). Total time for the Extrema Skeleton method however depends predominantly on the fraction of isosurface cells because polygonization is a time consuming operation. Figure 4.5 graphically represents the relationship between the performance of Extrema Skeleton Method, the dataset characteristics (fraction of seed cells), and the isovalues being searched (fraction of isosurface cells). For the datasets tested, performance of the Extrema Skeleton method was recorded by determining how much faster the Extrema

Skeleton method was compared to the Marching Cubes method according to the formula:

$$\% \text{ Faster} = (T_{MC} - T_{ES})/T_{MC} * 100$$

Performance was then plotted against 'Dataset Parameter' where,

Dataset Parameter = (Fraction of Isosurface cells) + 0.08 x (Fraction of Seed cells)

Dataset Parameter was chosen in the form shown above for the following reasons:

a) Performance primarily depends on the fraction of isosurface cells.
b) The constant "0.08" in the equation was chosen because the average time to process a non-isosurface cell was found to be approximately 8% of the average time taken to process an isosurface cell.

It was found that as the value of Dataset Parameter increased, performance of the Extrema Skeleton method decreased. It can be concluded from Figure 4.5 that, for the simple datasets tested, the Extrema Skeleton method was at least 31% faster. For the remaining datasets tested, the Extrema Skeleton method was marginally (<5%) faster.

The number of polygons generated when ten isosurfaces were extracted is shown in Table 4.6 for each dataset. In every case, the number of polygons generated was equal for both methods. This indicates that the Extrema Skeleton method gives the same end result as the Marching Cubes method by searching fewer cells.

In comparing the timing results of both methods, preprocessing time for the Extrema Skeleton method was not considered. This is because preprocessing time is independent of the isovalues searched and also independent of how many isosurfaces are

extracted. Preprocessing time for a given dataset depends only on its size. If preprocessing time were included in the the timing results when comparing the two methods, performance of the Extrema Skeleton method could be bettered by simply extracting more isosurfaces. This point is illustrated in Figure 4.6. Timing results were compared by extracting 10 isosurfaces and 25 isosurfaces from the dataset SomeIso.vtk. Results show that for 10 isosurfaces, Marching Cubes was faster while for 25 isosurfaces, the Extrema Skeleton Method was faster. For both cases, preprocessing time was the same.

4.4 COMPARING RESULTS WITH PREVIOUS WORK

Itoh [10] compared the performance of the Extrema Skeleton method with other algorithms that require preprocessing. Results of his study indicated that the preprocessing time for the Extrema Skeleton method was in most cases less than the preprocessing time for other methods. He also found that the preprocessing time for the Extrema Skeleton method increased linearly with the size of the dataset. His observations match the findings of the current study. Itoh also found that the time taken by the Extrema Skeleton method depends on the size of the dataset and the number of extremum points, but primarily on the number of isosurface cells. This is consistent with the observations of the current study where the performance of the Extrema Skeleton method was observed to depend on the fraction of isosurface cells and the fraction of seed cells (depends on extremum points and dataset size) in the dataset. Further

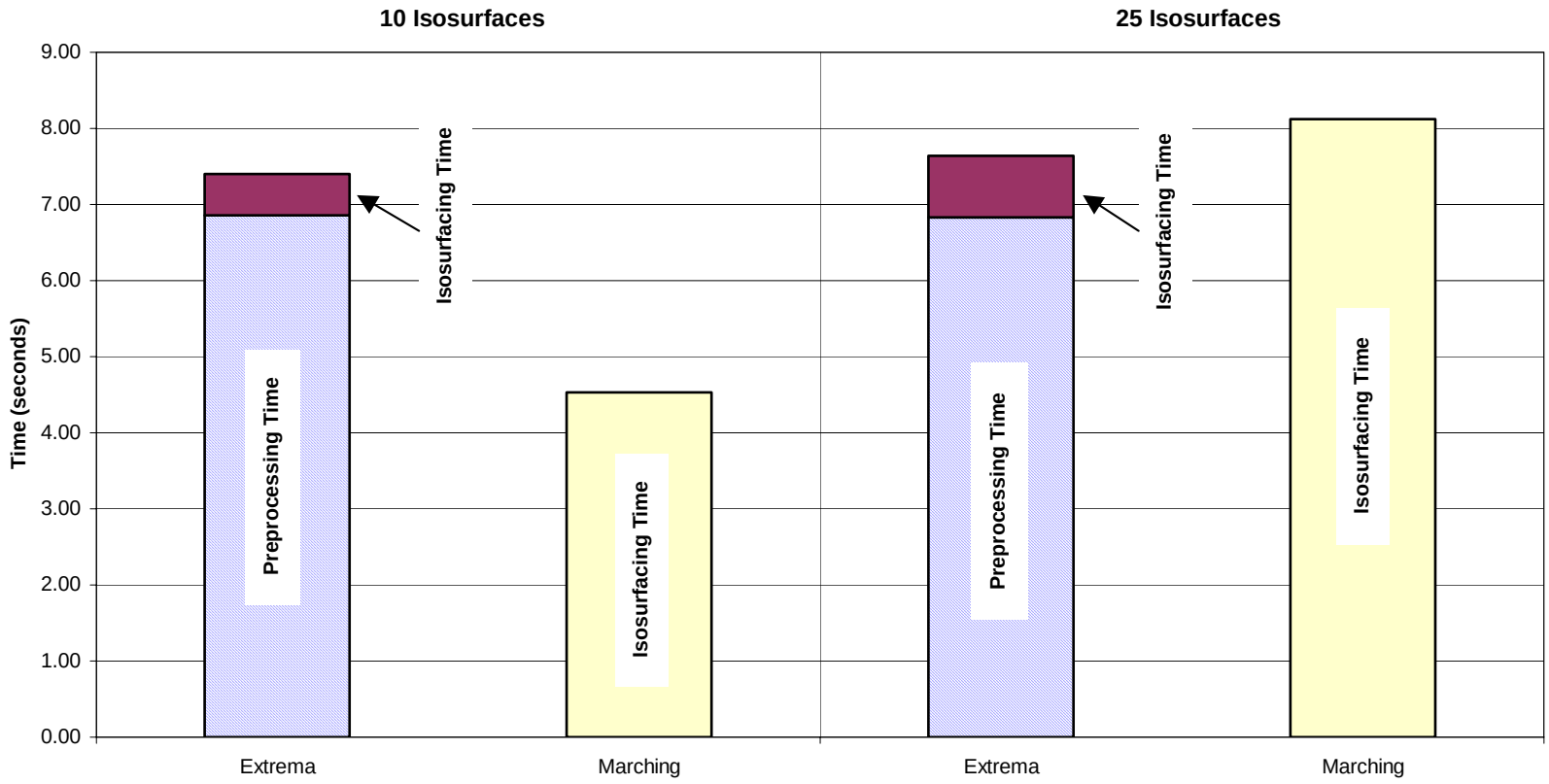


Figure 4.6: Amortization of Preprocessing Time for the Extrema Skeleton Method

comparison of the results is restricted by the differences between the type of datasets tested, and type of dataset cells. The type of datasets tested in the current work represented a structured grid. Datasets tested by Itoh were all unstructured grids. Similarly, while the datasets tested in the current work comprised of hexahedral cells, datasets tested by Itoh comprised of tetrahedral cells.

4.5 CONCLUSIONS

As stated in the problem statement described in Section 1.2, the objective of this thesis was to investigate the hypothesis that the Extrema Skeleton algorithm will decrease the time taken to extract isosurfaces from a given dataset, when compared to the Marching Cubes algorithm in the VTK package. The intent was to decrease the computing time by reducing the number of cells searched. This hypothesis was found to be true for most but not all of the datasets tested. Conclusions from this study are listed below:

1. Preprocessing time for the Extrema Skeleton method is linearly proportional to the size of the dataset.
2. For simple datasets with less than 10% isosurface cells and complex datasets with less than 5% isosurface cells, the Extrema Skeleton method is at least 10% faster than the Marching Cubes method.
3. For complex datasets with greater than 10% isosurface cells and any dataset with greater than 15% isosurface cells, the speedup gained by the Extrema Skeleton method is insignificant.

4. Real data is usually not based on smooth functions. Therefore, users of VTK are expected to encounter datasets that have at least 30% seed cells. Users are also very likely to deal with datasets having less than 10% isosurface cells. Therefore, implementing the Extrema Skeleton method for the VTK software is worthwhile because VTK users deal with datasets for which the Extrema Skeleton method is significantly faster and also with datasets for which the Extrema Skeleton is marginally faster than the Marching Cubes method.

LIST OF REFERENCES

LIST OF REFERENCES

[1] W.E.Lorensen, and H.E.Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *ACM Computer Graphics*, Vol. 21, No. 4, pp. 163-170, July 1987.

[2] Doi, and A. Koide, "An Efficient Method of Triangulating Equivalued Surfaces by Using Tetrahedral Cells," *IEICE Transactions*, Vol. E74, No. 1, pp. 214-224, 1991.

[3] D. Speray, and S. Kennon, "Volume Probe: Interactive Data Exploration on Arbitrary Grids," *Computer Graphics*, Vol. 24, No. 5, pp. 5-12, 1990.

[4] J.Wilhelms, and A.Van Gelder, "Octrees for Faster Isosurface Generation," *ACM Transactions on Graphics*, Vol. 11, No. 3, pp. 201-227, July 1992.

[5] R.S.Gallagher, "Span Filtering: An Optimization Scheme for Volume Visualization of Large Finite Element Models," *IEEE Visualization '91*, pp. 68-74, 1991.

[6] H.W.Shen, C.D.Hansen, Y.Livnat, C.R.Johnson, "Isosurfacing in Span Space with Utmost Efficiency (ISSUE)," *Visualization '96 Conf. Proc.*, pp. 287-294, San Francisco, October 1996.

[7] M. Van Kreveld, R.Van Oostrum, C.Bajaj, V.Pascucci, and D.Schikore, "Contour Trees and Small Seed Sets for Isosurface Traversal," *In Proc. 13th ACM Symposium on Computational Geometry*, pp. 212-220, 1997.

[8] T. Itoh, and K. Koyamada, "Isosurface Generation by Using Extrema Graphs ," *IEEE Computer Society Press Reprint*, pp. 77-83, 1994.

[9] Y.Livnat, H.W.Shen, and C.R.Johnson, "Near Optimal Isosurface Extraction Algorithm Using the Span Space," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 2, No. 1, pp. 73-84, March 1996.

[10] T.Itoh, Y.Yamaguchi, and K.Koyamada, "Fast Isosurface Generation Using the Volume Thinning Algorithm," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 7, No. 1, pp. 32-46, January-March 2001.

[11] T. Itoh, and K. Koyamada, "Automatic Isosurface Propagation Using an Extrema Graph and Sorted Boundary Cell Lists," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 1, No. 4, pp. 319-327, December 1995.

[12] T.Itoh, Y.Yamaguchi, and K.Koyamada, "Volume Thinning for Automatic Isosurface Propagation," *IEEE Visualization '96 Conf. Proc.*, pp. 303-310, 1991.

[13] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno, "Speeding Up Isosurface Extraction Using Interval Trees," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 3, No. 2, pp. 158-170, April-June 1997.

[14] C.T.Howie, and E.H.Blake, "The Mesh Propagation Algorithm for Isosurface Construction," *Computer Graphics Forum (Eurographics)*, Vol. 13, No. 3, pp. C-65-74, 1994.

[15] W.Schroeder , K.M.Martin , W.E.Lorensen, "*The visualization toolkit (2nd ed.): an object-oriented approach to 3D graphics,"* Prentice-Hall, Inc., Upper Saddle River, NJ, 1998.

[16] S.Rottger, M.Kraus, and T.Ertl, "Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection," *In Proc. Visualization 2000, IEEE Computer Society Technical Committee on Computer Graphics*, pp. 109-116, 2000.

[17] J.W.Durkin, and J.F.Hughes, "Nonpolygonal Isosurface Rendering for Large Volume Datasets," *Proceedings of Visualization '94, IEEE*, pp. 293-300, 1994

[18] MathWorld.com, http://mathworld.wolfram.com/SaddlePoint.html, August 2003.

APPENDIX

APPENDIX A

Equations Used to Generate Datasets

The following equations were used to generate the datasets Ellipsoid.vtk, Paraboloid.vtk, and Hyperboloid.vtk:

Ellipsoid.vtk

$$f(x, y, z) = \frac{x^2}{64} + \frac{y^2}{64} + \frac{z^2}{25} - 1$$

f(x,y,z) was assigned as the scalar value at the point (x,y,z).

Data was generated in the range (0,0,0) to (10,4,8) using a step size of 0.25. This range will yield the section of the ellipsoid shown in Figure B-1.

Hyperboloid.vtk

$$f(x, y, z) = \frac{x^2}{64} + \frac{y^2}{64} - \frac{z^2}{25} - 1$$

f(x,y,z) is assigned as the scalar value at the point (x,y,z).

Data was generated in the range (0,0,0) to (6,9,8) using a step size of 0.2. This range will yield the section of the hyperboloid shown in Figure B-2.

Paraboloid.vtk

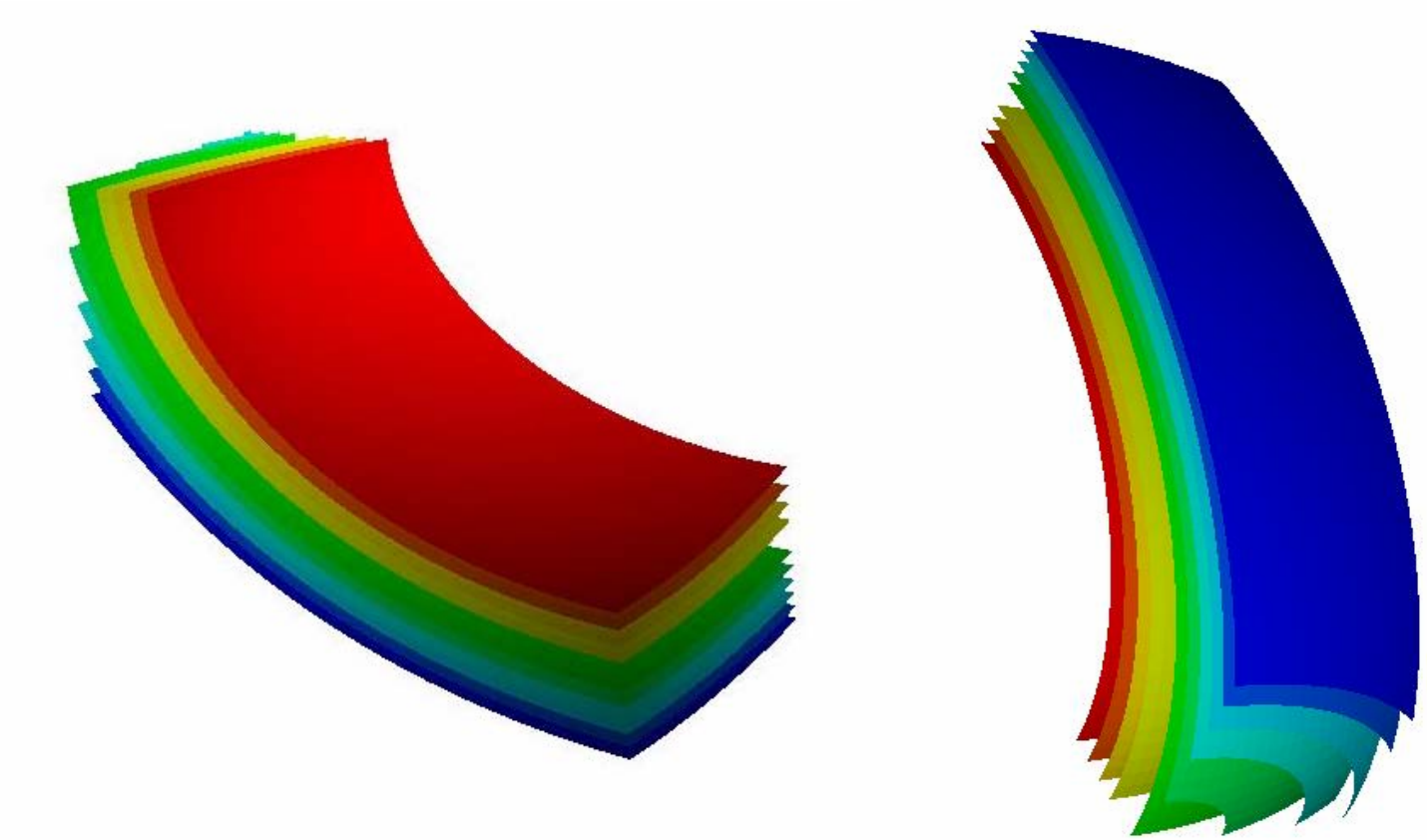
$$f(x, y, z) = 2\left(x^2 + y^2\right) - z$$

f(x,y,z) is assigned as the scalar value at the point (x,y,z).

Data was generated in the range (0,0,0) to (1,4,2) using a step size of 0.05. This range will yield the section of the paraboloid shown in Figure B-3.

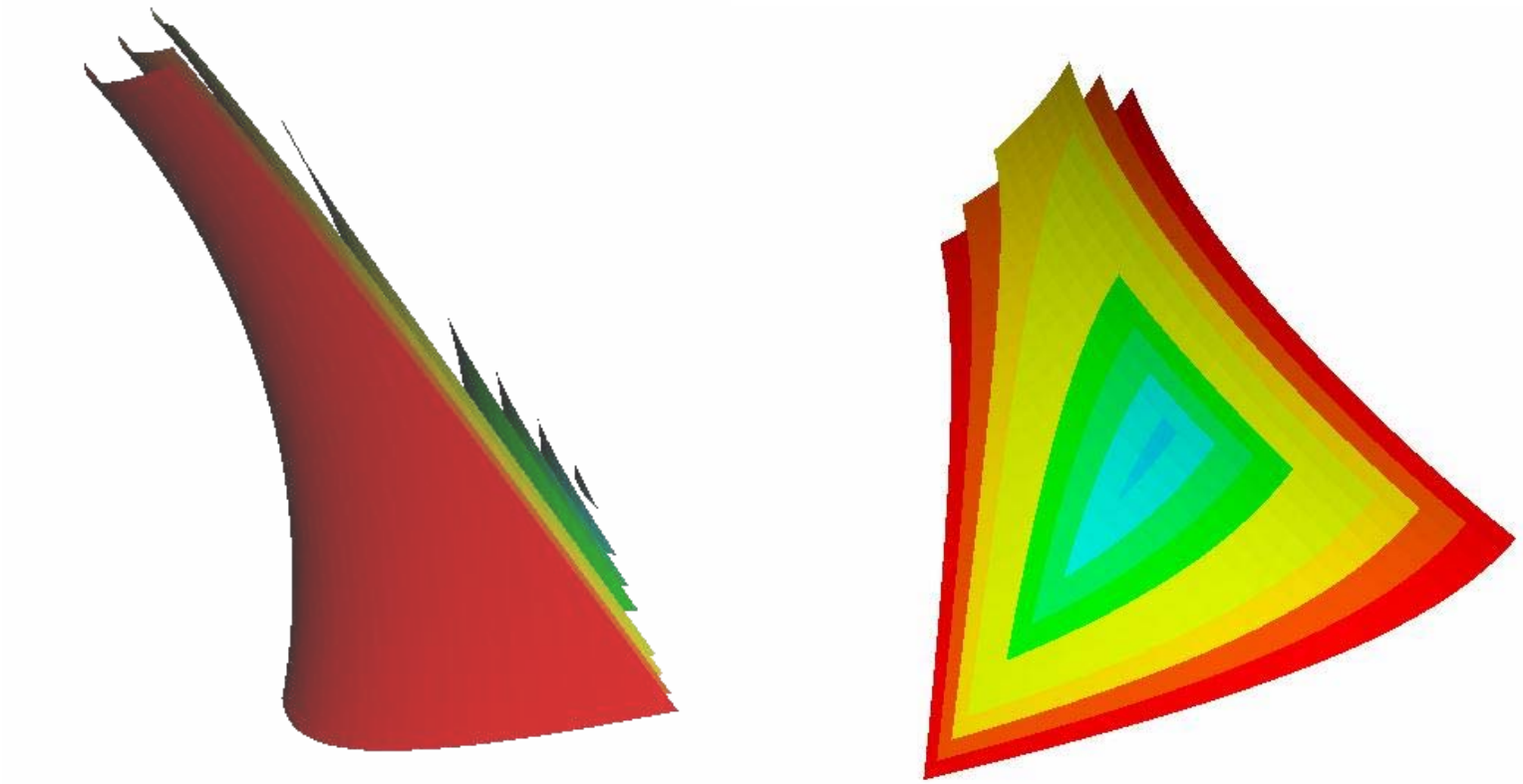
APPENDIX B

Isosurfaces Extracted From the Datasets Tested



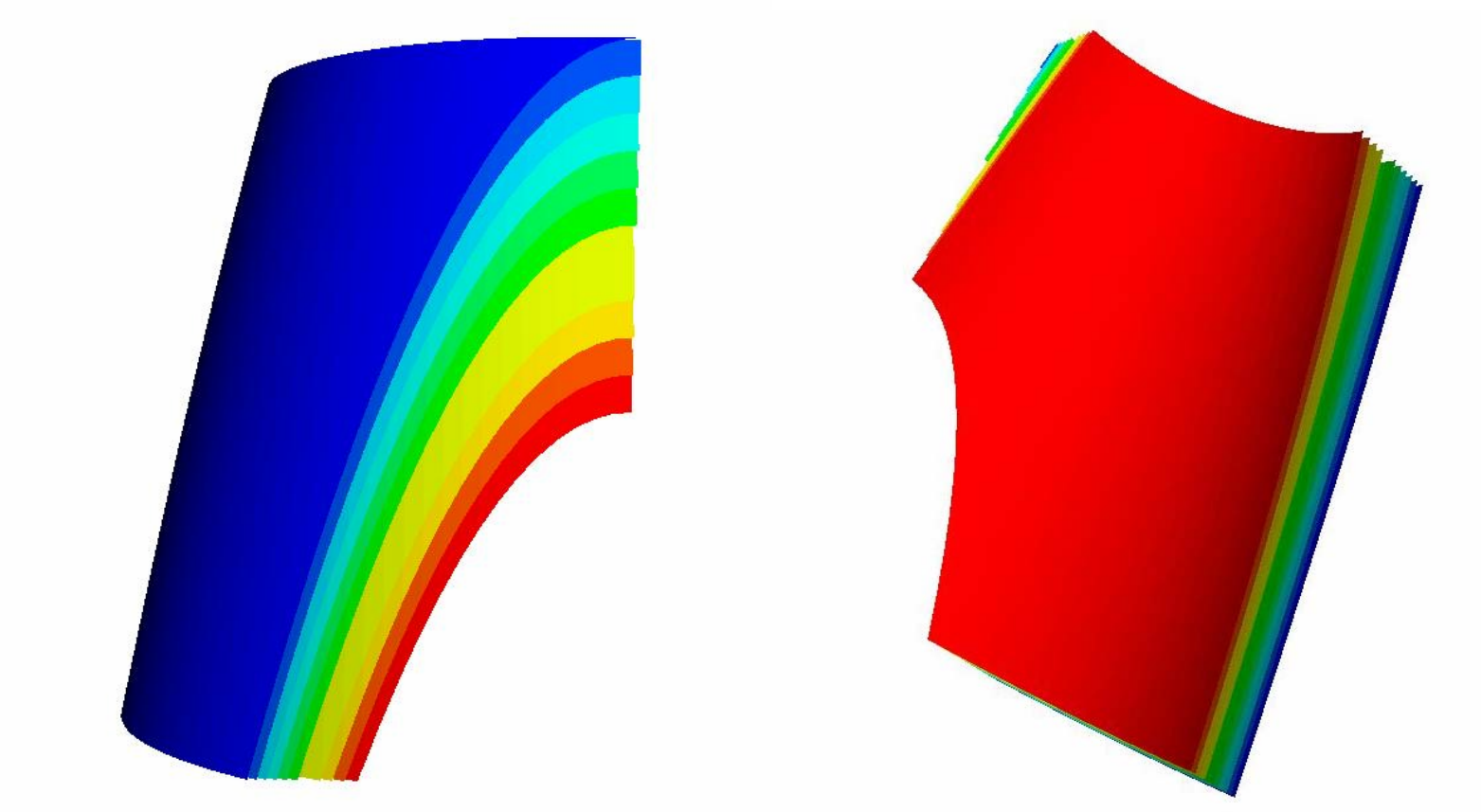
View 1 **View 2**

Figure B-1: Ten Isosurfaces Extracted from Ellipsoid.vtk



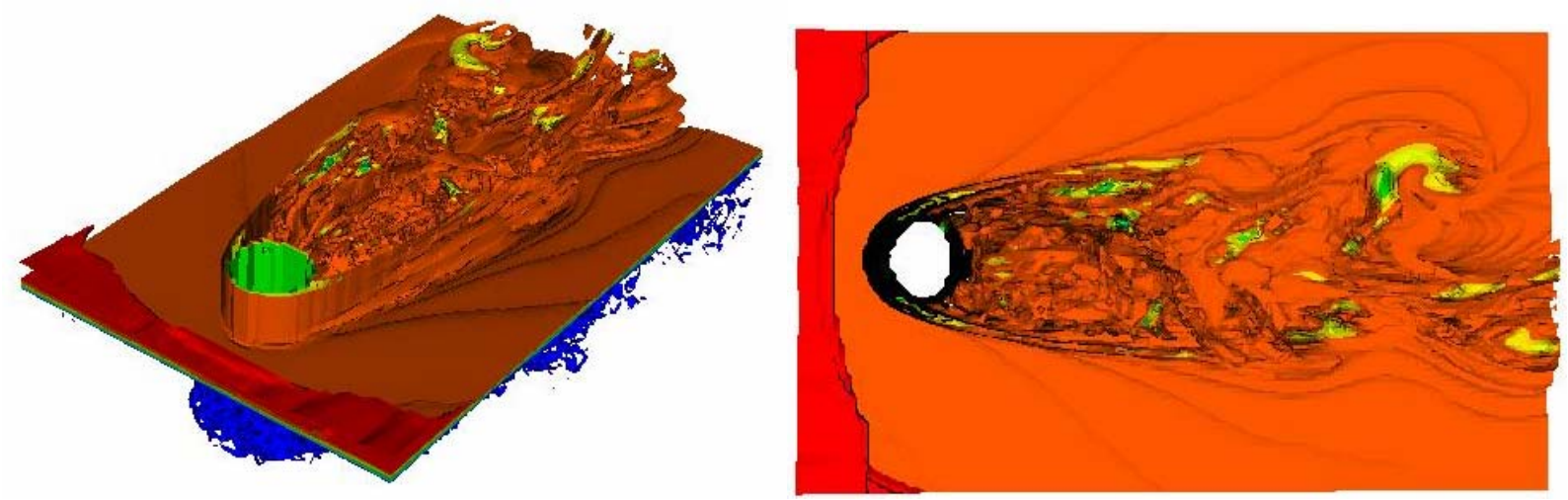
View 1 **View 2**

Figure B-2: Ten Isosurfaces Extracted from Hyperboloid.vtk



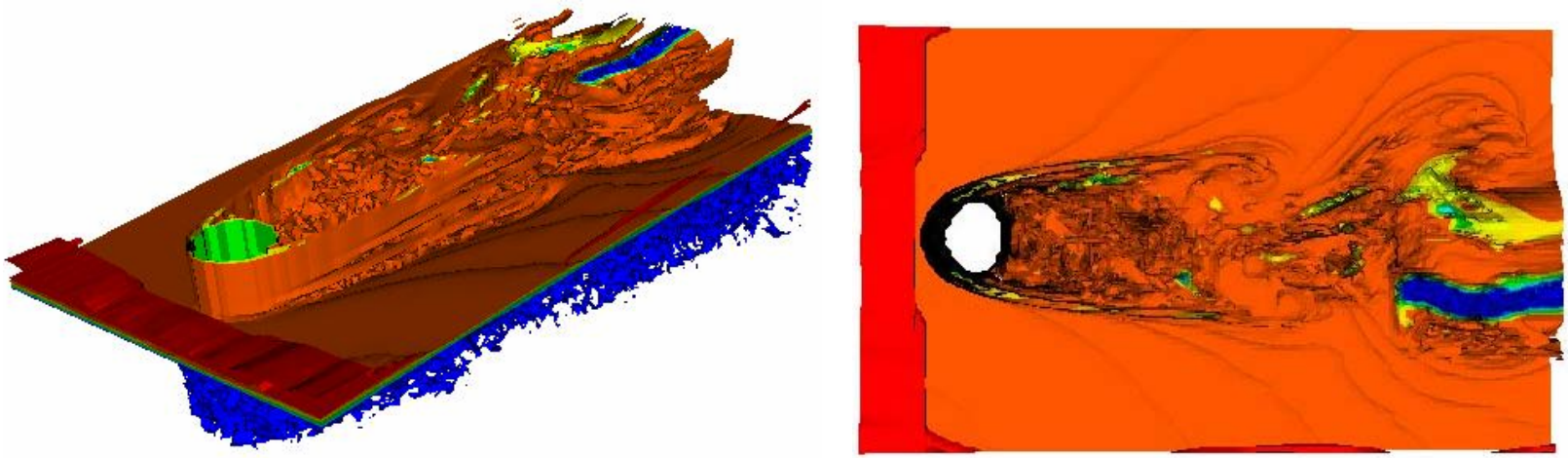
View 1 **View 2**

Figure B-3: Ten Isosurfaces Extracted from Paraboloid.vtk



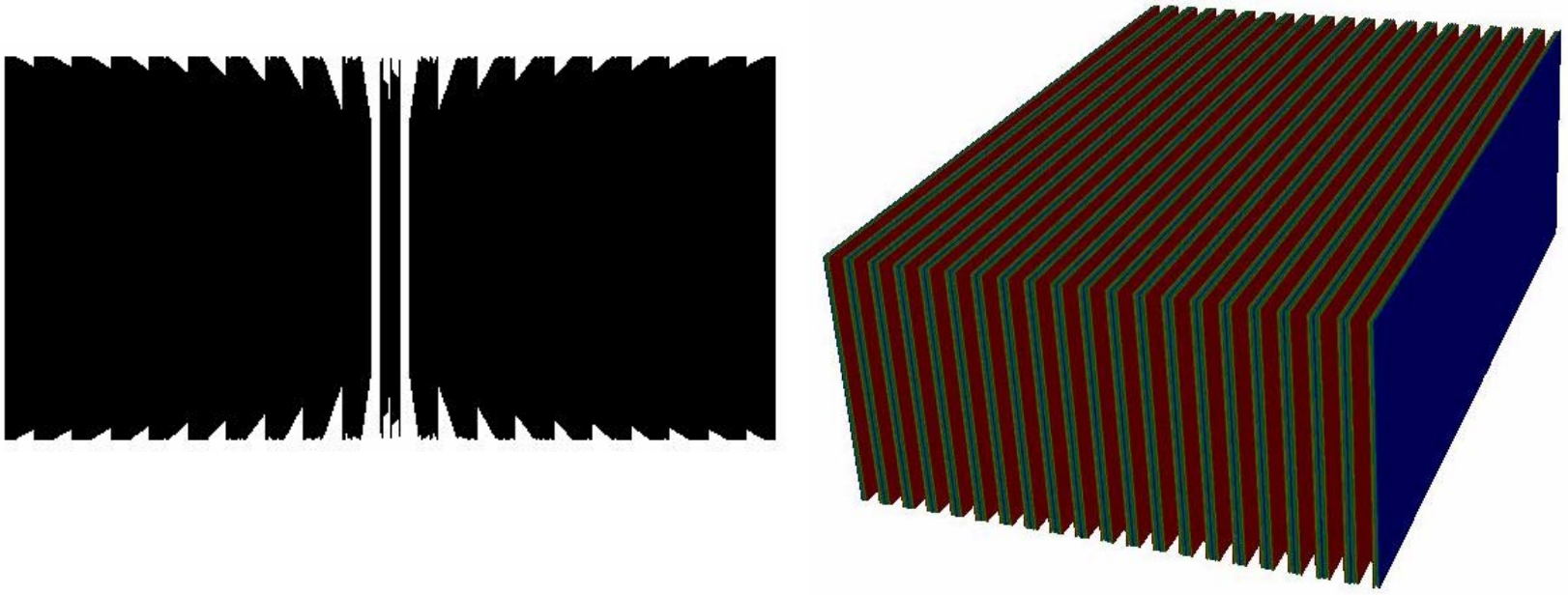
View 1 **View 2**

Figure B-4: Ten Isosurfaces Extracted from FlowFrame1.vtk



View 1 **View 2**

Figure B-5: Ten Isosurfaces Extracted from FlowFrame20.vtk



View 1 **View 2**

Figure B-6: Ten Isosurfaces Extracted from AllIso.vtk

APPENDIX C

Visualization of Regular Objects

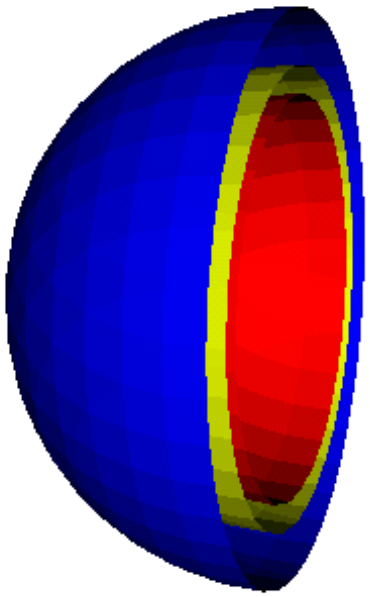


Figure C-1: Isosurfaces Extracted from Data Generated Using Equation of a Sphere

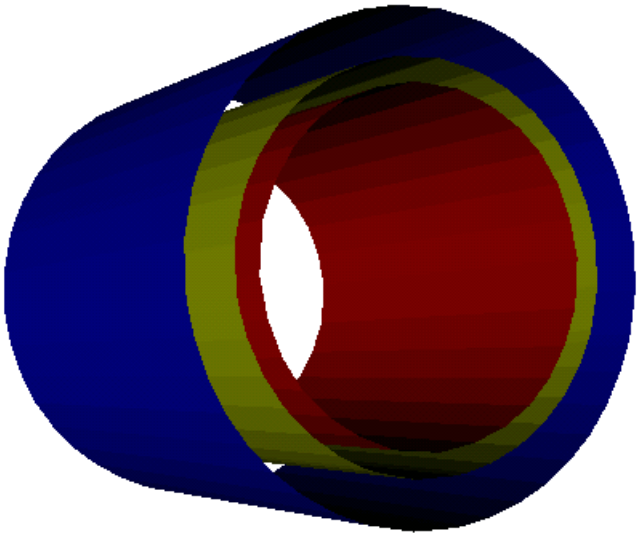


Figure C-2: Isosurfaces Extracted from Data Generated Using Equation of a Cylinder

VITA

Subha Mahaadevan was born in Chennai, India on April 29, 1976. She attended schools in Chennai and Pondicherry, India. She received her Bachelor of Science degree in Mathematics from the University of Madras in 1996, and her Master of Science degree in Mathematics from Pondicherry (Central) University in 1998. She got her Diploma in Software Engineering and Systems Management in 1999 from the National Institute of Information Technology, Chennai, before coming to the University of Tennessee Space Institute to pursue Master of Science in Computer Science. She is currently employed at the Center for Laser Applications, UTSI, as Senior IT Technologist-II.