A Hybrid Deformable Model 3-D Segmentation Algorithm

by

Varouj A. Chitilian

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science and Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1999

Author .....

Electrical Engineering and Computer Science
May19, 1999

Certified by .....

W. EricL. Grimson
Professor of Computer Science and Egineering
Thesis Supervisor

Accepted by .....

Arthur C. Smith
Professor of Electrical Engineering and Computer Science
Chairman, Department Committee on Graduate Theses

Acknowledgments

I would like to thank Prof. Grimson for his courage in accepting to be my thesis advisor, and also David Gering and Polina Golland of the Medical Vision Group at the Artificial Intelligence Laboratory at MIT, for their kind help and support.
I would also like to thank Dr. Clay Spence and Dr. Paul Sajda of Sarnoff Corporation for their time during long discussions concerning this thesis, and for their ideas and input.
Last, but not least, I would like to thank my family for their support through the toughest times of my MIT career. Thank you Mom, Dad, and Hovig for your kindness.

A Hybrid Deformable Model 3D Segmentation Algorithm

by

Varouj A. Chitilian

Submitted to the Department of Electrical Engineering on May 21, 1999, in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science and Masters of Engineering in Electrical Engineering and Computer Science

The thesis project describes the design and implementation of a hybrid deformable model 3D segmentation algorithm. The algorithm is based on two well-known and respected algorithms: the Topologically Adaptable Surface algorithm by McInerney and Terzopoulos, and the front propagation: level set algorithm by Malladi, Sethian and Vemuri. The thesis attempts to bring together the good qualities of each algorithm in the creation of a simple to understand, simple to implement segmentation algorithm. Testing on the full algorithm was limited. However, the hybrid algorithm performed very well on the test data used.

Thesis Supervisor: Prof. W. Eric Grimson
Title: Professor of Computer Science and Engineering

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 The Fundamental Problem

The fundamental problem towards which this thesis will be directed is segmentation. Segmentation is defined as the process of "dividing the image into regions that correspond to structural units in the scene or distinguish objects of interest."[1].

The process in effect reduces an image into some form of relevant information. Therefore the algorithms presented in this thesis all attempt to extract a description of the surface of an object of interest from an image.

The problem of segmenting an image does not have an exact solution. The best once can hope for is an accurate approximation. The problem of segmentation also lacks a general algorithm that can be used in all applications. Therefore the field has been developed as an application specific field. In other words, methods and algorithms developed which try to solve the problem of segmentation are in the most part dedicated to specific applications, and make a priori assumptions that apply only to the application in which segmentation is being used.

A large application area where segmentation can be used, and is in some cases necessary, is the field of medical image analysis. The algorithms outlined and proposed in this thesis are all methods developed for this field. They belong to a set of algorithms collectively known as "deformable model" algorithms.

This thesis will describe the theory behind, and the implementation of a novel three-dimensional deformable model algorithm, which is a hybrid of two existing, well-known algorithms.

The algorithm is useful because it is simpler both theoretically and practically, in terms of implementation in three dimensions, than the two algorithms it is derived from, but still provides accurate results with minimal supervision.

1.2 Application- Breast Cancer Classification

The process of segmentation has been put to extensive use in medical areas. One of the more recent of these areas is tumor classification in the breast area of females from MR images. Studies have shown that there is a strong correlation between the roughness of the surface of a tumor and whether the tumor is a carcenoma (a malignant tumor) or a fibroadenoma (a benign tumor that is difficult to distinguish from carcenomas using other more conventional techniques). Typically, a carcinoma exhibits irregular borders, varying from only minimal irregularity to gross spiculation. In contrast a fibroadenoma has regular, often lobulated borders [2]. Figure 1a shows a picture of a benign tumor; figure 1b shows a picture of a cancerous tumor.

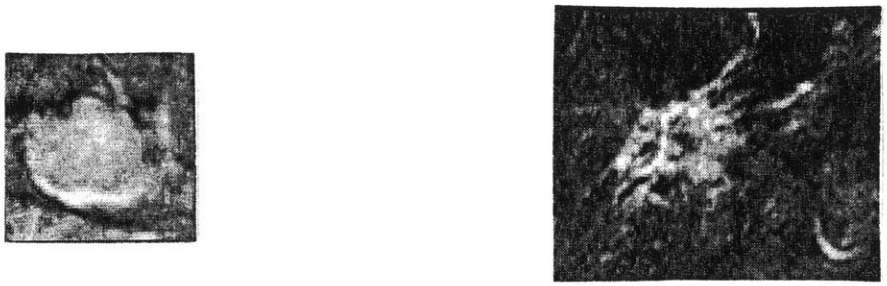


Figure 1.1: a) A 2-D slice of a fibroadenoma. Notice the relatively smooth border. b) A 2-D slice of a carcenoma. Notice the protrusions and rough boundary details.

The goal of using a segmentation algorithm would be to render the process of determining if a patient should be diagnosed with breast cancer, more automatic, and less expensive. Currently a specialist must look at MR images individually and rank the roughness of the surface of the tumor on a scale of 1 to 5[2]. If one can create a good segmentation algorithm and extract some metric from the output surface that accurately measures roughness, one could automate the process and perform this type of diagnosis without supervision.

Because the process must be automated, a useful segmentation algorithm should be robust towards changes of shape of the object being segmented, and also should not be affected by initial conditions used to initialize the algorithm.

Due to the importance of this diagnosis, the segmentation algorithm must give a very accurate representation of what appears to be the boundary of the tumor, so that the metric can in turn approximate the roughness accurately; a general description of the surface will not suffice. Three-dimensional segmentation is important in this respect because one can imagine an MR image respresenting a slice through the tumor that shows smooth boundaries, when in fact the tumor as a whole has a rough surface.

Chapter 2

Previous Work In the Field

2.1 Deformable Model Algorithms

Segmentation is different than simple edge detection because it assumes certain properties about the object that is being segmented. The assuptions that are made about the object become constraints to what the segmentation algorithm can output. Often they are these assumptions that define what type of segmentation algorithm should be used.

Deformable model algorithms are powerful, physics-based algorithms for recognizing non-rigid surfaces from their images [3]. It can be assumed that these algorithms are well suited for breast tumor classification and medical image analysis in general, because non-rigidity is a fundamental characteristic of soft tissues in the human body and other biological structures.

There has already been much work done in the development of deformable model algorithms, but as Ajit Singh comments in the preface of his book Deformable Models in Medical Image Analysis: "There is a need to juxtapose various techniques based on deformable models, compare their disadvantages and pitfalls, bring together the best of all approaches, and build clinically oriented systems for validation and eventual deployment in routine clinical practice."[3]

This report proposes just that: an algorithm based on two widespread and generally accepted algorithms in the field that have their advantages and disadvantages. My hope in combining the two algorithms is that I will be able to generate an algorithm that can capitalize on the strengths and get rid of the weaknesses of each algorithm.

The two algorithms that my algorithm will be based on are the "Medical Image Segmentation using Topologically adaptable Surfaces" by Tim McInerney and Demetri Terzopoulos [4], and "Shape Modeling with Front Propagation: A Level Set Approach" by Ravikanth Malladi, James A. Sethian, and Baba C. Vemuri [5].

2.2 Topologically Adaptable Surfaces

The topologically adaptable surface algorithm for segmentation is a direct extension of the topologically adaptable snake algorithm, which in turn has its roots in one of the first major deformable model algorithms published, namely "Snakes: Active Contour Models" by Michael Kass, Andrew Witkin, and Demetri Terzopoulos, which appeared in the International Journal of Computer Vision in 1987 [6].

The Snakes algorithm uses energy-minimizing splines guided by external constraint forces and influenced by image forces that pull it toward features such as lines and edges. The internal spline forces serve to impose a piecewise smoothness constraint. The image forces push the splines (referred to as snakes) toward salient image features like lines, edges, and subjective contours. The external constraint forces are responsible for putting the snake near the desired local minima. So if a snake was parametrically defined as:

$$V(s) = (x(s), y(s)) \tag{2.1}$$

Then it's energy functional would be:

$$E = \int_s E_{snake}(v(s))ds$$
$$= \int_s (E_{int}(v(s)) + E_{image}(v(s)) + E_{con}(v(s)))ds \tag{2.2}$$

The algorithm would attempt to minimize this value for each snake around the contour by picking recursively better v(s)'s.

If we look at each term closely we can appreciate the physics behind the model.

$$Eint = \frac{\left(\alpha(s)\left|\frac{d}{ds}v(s)\right|^2 + \beta(s)\left|\frac{d^2}{ds^2}v(s)\right|^2\right)}{2} \tag{2.3}$$

The internal energy of the spline is composed of a first order term that controls how much the spline acts like a membrane, and a second order term which determines how much the spline acts like a thin-plate. This type of spline is referred to as a controlled continuity spline.

$$E_{image} = -(G_\sigma \bullet |\nabla I|)^2 \tag{2.4}$$

This is just a simple image energy functional but it gives an idea of what this term is meant to do. In equation 2.4, G_σ reffers to a Gaussian filter with standard deviation σ, and $|\ I|$ simply represents the magnitude of the gradient of the image. The symbol between them is a convolution symbol which is used to blur the gradient magnitude of the image.

Therefore, The larger the gradient of the blurred image, the smaller the energy. Hence high-gradient regions will tend to pull the splines toward them.

The $\mathbf{E_{con}}$ energy functional represents an initial constraint imposed on the snake by a user, and any other effect the user might have during the evolution of the contour [6].

The Snakes algorithm was a landmark algorithm in the field of deformable models. However, it has many problems. First, one has to initiate the contour or set of connected snakes close to the object one wishes to segment. This is because the spline's internal energy would prevent it from bending too much in order to reach an edge. This human factor made segmentation an interactive process with a user, which is not desirable in this case. Second, the internal energy of the splines made finding sharp features, for example protrusions, in objects very difficult: the more the spline would have to bend to fit these sharp features the larger the internal energy would get.

The 1991 paper by Laurent D. Cohen entitled "On Active Contour Models and Balloons", solved the first of these problems [7]. Cohen introduced an additional force which made the curve behave like a balloon which is inflated. The initial curve no longer needed to be close to the solution to converge.

The second of the problems was not solved until 1995 when McInerney and Terzopoulos introduced "Topologically Adaptable Snakes" or T-snakes [8]. They developed a simple algorithm, which they could use to reparameterize their contour at certain points to include more splines, so that sharp features could be accomodated, without great internal energy increases. The same algorithm that allowed them to reparameterize also gave them the power to perform topological transformations, which are necessary for example when the contour collides with itself, or must change itself topologically by joining with or separating from other contours. In effect, the T-snakes extended the geometric and topological adaptability of traditional snakes.

The algorithms used in the T-snakes paper, to accommodate reparameterization and topological transformations, rely on the principles of a simplicial decomposition of space. In a simplicial decomposition, space is partitioned into cells defined by open simplicies, where an ***n-simplex*** is the simplest geometrical object of dimension n. The simplest simplicial decomposition of Euclidean space $\boldsymbol{R^n}$ can always be constructed by subdividing space using a uniform cubic grid, and subdividing each cube into $\boldsymbol{n!}$ simplicies.

Reparameterization of the T-snake is accomplished by superimposing a simplicial grid of arbitrary scale on the contour, and re-representing the nodes of each spline by where the contour crosses a given cell. If one chooses a scale small enough the original contour will be represented by a set of smaller splines, that will give more geometric flexibility to the new contour.

Simplicial decomposition also gives one an easy way of performing topological transformations. As mentioned earlier, when a snake collides with itself or with another snake, or when a snake breaks into two or more parts, a topological transformation must take place. When a contour is simple, and a simplicial grid is superimposed, each simplex cell will contain exactly one spline, that will connect nodes on different edges. However, when a contour intersects a simplex cell more than twice, this is a sign that the contour is not simple and a topological transformation is necessary. All non-simple contours can be simplified by performing operations on only the boundary cells, these are the simplex cells that have some vertices inside the contour and other vertices outside the contour. Because we are using a simplicial decomposition there is always a linear entity that can traverse the cell and separate the vertices of the cell which are inside the contour from those which are outside. The topological transformation algorithm therefore when applied to a boundary cell with more than two cell intersections, simplifies the contour by joining the two points of intersection on different edges of the cell which separate the inside vertices from the

outside ones, and are closest to the outside vertices [8]. The chapter on implementation notes will go further into why a simplicial decomposition is useful in performing topological transformations and also provide some intuition as to what is actually happening during a topological transformation.

The topologically adaptable surface algorithm is simply a 3D extension of the T-snake idea. The simplicial decomposition now splits each cube into 6 tetrahedra. The surface is reparameterized using the intersection of the surface with the edges of the simplex cells. Topological transformations are also performed in the same way as in T-snakes [4].

There are still some disadvantages to the T-surface algorithm. It remains an interactive segmentation algorithm, which depends heavily on user-provided information about when to superimpose a simplicial grid, and thus reparameterize, and how coarse or fine the simplicial cells should be.

The accuracy of this algorithm is also an issue. Because of the very detailed nature of a tumor's borders, splines in general are not very helpful in the segmentation process.

Also, it is necessary for a user to have good knowledge and intuition of splines, which makes the algorithm theoretically more sophisticated and complex.

2.3 Propagation of Fronts: A level set approach

In 1995, Malladi, Sethian, and Vemuri suggested another important deformable model algorithm in a paper entitled, "Shape Modeling with Front Propagation: A Level Set Approach" [5]. This paper proposed using a front, a closed, non-intersecting, hypersurface flowing along its gradient field with speed that depends on the curvature, and syn-

thesizing a speed term from the image to stop the front in the vicinity of the object boundaries.

Malladi et al. represented the front in R^N as a level set $\gamma(t)$ of a function $\Psi(x,t)$, where x is an N dimensional position vector, such that:

$$\gamma(t) = \{x_0 : \psi(x_o, t) = 0\} \tag{2.5}$$

With the assumption that $\gamma(t)$ will always move such that every point on $\gamma(t_n)$ will travel in the normal direction to $\gamma(t)$ at their location, Malladi et al. were able to develop a time evolution law that described the motion of the function $\Psi(x,t)$ with a "Hamilton-Jacobi" type differential equation:

$$\Psi_t + \boldsymbol{F} \cdot |\nabla \Psi| = 0 \tag{2.6}$$

where **F** is a function representing the speed of the hypersurface at any given point. The motivation behind using the level set method now becomes clear. In a Hamilton-Jacobi type equation such as this, the function Ψ will remain a function so long as **F** is a smooth function. However, the level surface $\{\Psi = 0\}$ which represents the front may change topology, break, merge and form sharp corners as Ψ evolves.

In order for the segmentation to work, Malladi et al. were able to identify three major factors that could influence the speed. A constant factor, $\mathbf{F_A}$, that they call the advection term, which represents an expansionary force similar to the balloon force; a facture due to geometry, $\mathbf{F_G}$, Malladi et al. used curvature dependent term to represent this; and of course an image dependent term, $\mathbf{k_I}$. So the final speed term they used was:

$$F = K_I \cdot (F_A + F_G) \tag{2.7}$$

Malladi et al. further developed the terms as follows. They made $\mathbf{F_G}$ a curvature dependent term of the form 1 - εK, where K represents the curvature of the contour at the given point. They made $\mathbf{k_I}$ be an inverse exponential dependent on the gaussian of the gradient of the image at the given point. And finally they made $\mathbf{F_A}$ simply a constant factor [5].

They then went on to describe how to calculate image dependent speed terms for all points on Ψ, regardless of whether they are on the image plane or not. This lead to a development of a global speed term that could be applied generally to all points of Ψ.

Once the algorithm has been initialized with a γ(t=0), it simply follows the general rule for evolution at each time step. The front expands because of the advection speed term, stays smooth because of the curvature dependent term, and slows down exponentially when it hits an image feature that has a high image gradient. Topological transformations are taken care of automatically by the evolution of Ψ.

Although this algorithmm is very accurate and automatic, it is both theoretically and practically quite complex.

Chapter 3

Design of a Hybrid Algorithm

3.1 Dissecting the Deformable Model Segmentation Process

There are four key components that are present in all deformable model segmentation algorithms. In fact, these components are necessary to ensure accurate results.

Any deformable model algorithm must contain a good model representation, some set of rules or laws by which this model can evolve, a consistent method of performing topological transformations, and some criteria that allows the algorithm to determine an optimal model.

The following sections will analyze the importance of each of these components to the accuracy of the overall algorithm, and will elaborate on the different components selected for the hybrid algorithm presented in this thesis.

3.2 Model Representation

A model in three dimensions is represented by a set of points which lie on the surface of the model. A deformable model algorithm samples image data at the surface point locations of the model at iteration i to determine how to deform itself to better fit the image data at iteration $i+1$. Since very little can be assumed about the shape of the object in the image data, a well-chosen set of surface points are needed so that when the samples of the image data are taken, they provide the algorithm with accurate information regarding what is happening in a given region of the image at a given time. There are two general criteria for a good representation of a model. It must contain enough points, and the points must be spread out evenly enough. These two criteria ensure that higher frequency components

in the image data are not overlooked by the algorithm. Even if we do obtain a good representation of a model at a given time, complications arise when we evolve the model. The evolution of the model leads to an increase in its surface area. However, the number of model points remains the same, and therefore the model representation is less dense and the algorithm cannot as accurately interpret image features along the surface of the model. So, one can easily see that a consistent algorithm is required to find a good model representation after each iteration (ie. evolution of the previous model). Such a reparameterization algorithm using a simplicial decomposition of space is presented in the topologically adaptable snake paper, and this type of algorithm is what was used in the hybrid algorithm presented in this thesis. According to the topologically adaptable snakes paper, the simplicial grid used for reparameterization leads to an evenly distributed set of points on the surface of the model.

The number of points in the reparameterized surface can also be controlled with an added feature which allows the user to control the coarseness of the simplicial grid used in the reparameterization process. Therefore as the surface gets larger, one can go to coarser and coarser resolutions, to conserve memory or time. Figures 3.1 and 3.2 show the reparameterization process of the algorithm. In figure 3.1 the first image represents the object inputted into the process. It is a cube-like surface described by 8 points. When it is reparameterized with a coarseness factor of 1, the second image in the set is obtained. The surface is now represented by approximately 2000 points distributed evenly along it.Notice the effect of the reparameterization in introducting quantization effects. The edges of the surface are now less sharp. The third image shows the input surface when reparameterized using a coarseness factor of 2. Notice that the quantization effects on this third image are more pronounced. This is the result of using less points to describe the same surface, again sharp features like the edges are where we notice the effects most.

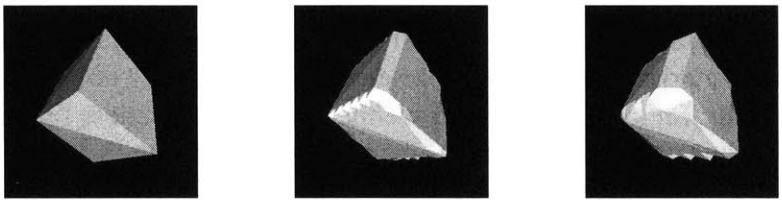


Figure 3.1: The first image on the left shows a surface described manually with eight points. The reparameterization algorithm can now operate on it to produce a similar surface with approximately 2000 points distributed evenly along the surface. The middle image shows the result of operating on the first image. To conserve memory in the final image the reparameterization algorithm has been implemented with a coarseness factor of 2. Notice how there are more reparameterization artifacts present and the surface is not as faithful to the original image.

In figure 3.2, we have reparameterized a larger input image using a coarseness factor of 2 in the second image and a factor of 3 in the third image. As opposed to the figure 3.1 where the input surface had a volume of about 8000, the input into figure 3.2 has a volume of about 27000. We can see that the artifacts due to the quantization have lessened drastically. This is due to the fact that the maximum error at any given coarsenss factor is constant, so that if the dimensions of the surface are larger this maximum error looks smaller in comparison. Even if we apply the reparameterization to the original image with a coarseness factor of 3, as in the last image, the surface is still relatively faithful to the original.

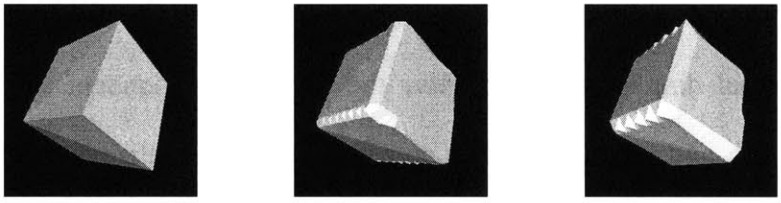


Figure 3.2: This set of images show that the artifacts become less and less significant as the surface grows larger. This is due to the fact that at any given coarseness factor, the

maximum error in reparameterization is constant. Therefore, for example in this set, if we increase the size of the surface by a factor of 3 from that in Figure 3.1, to obtain the first image, and we reparameterize again with a coarseness factor of 2, we obtain the middle image which is very faithful to the original. If we then go to a coarseness factor of three, we obtain the third image, which is still better than the second image in Figure 3.1, even though the courseness factor there was 1.

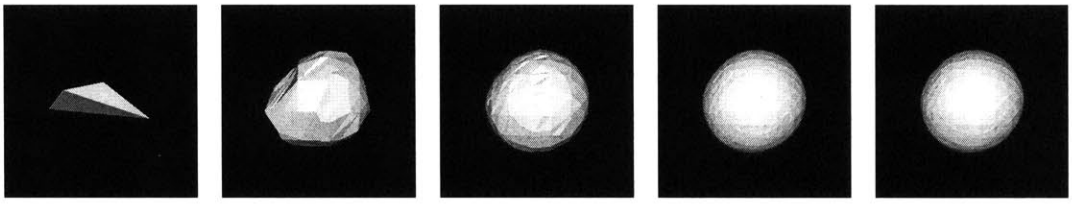
3.3 Model Evolution

Once we have a well represented surface, we cannot simply evolve this surface randomly in time. But rather the model must evolve in such a way that it gets closer and closer to the surface of the object being segmented. To come up with good laws of evolution for the model, assumptions about the objects being segmented must be considered.

In order to allow the hybrid algorithm to be able to segment as general a class of objects as possible accurately, the hybrid algorithm makes only very few simplifying assumptions about objects being segmented. These assumptions are very similar to those seen in various front propagation and snakes algorithms. And so the hybrid algorithm uses the law of evolution described in eq. 2.7. At each iteration, each point on the surface moves in its "approximated normal" (this approximation will be discussed in chapter 4) direction, with a magnitude that takes into consideration the magnitude of the image gradient of the gaussian of the image at the point, and the "approximated curvature" (again this approximation will be disussed in chapter 4) of the surface at the point.

Figure 3.3 shows the effects of evolving different surfaces with curvature dependent speed, in there normal directions. Notice how the curvature dependent speed tends to smooth the model out, and in the case of a) converts a small initialization volume into a sphere after 60 or so iterations. In b) one can see the smoothing effect more clearly. In this larger surface, we notice that the corners become rounded as the surface expands.

a)



b)

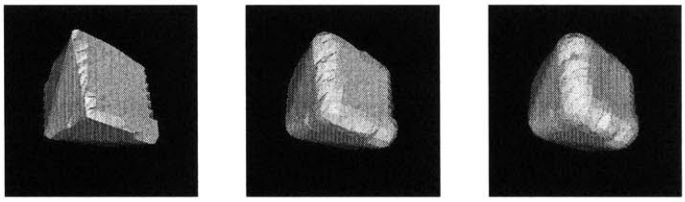


Figure 3.3: a) shows profound effect of curvature dependent speed on small volumes. Here a tetrahedra of volume four was used to initialize the program. Shown are the first, sixth, ninth and thirteenth iterations, by the end the tetrahedra has turned into a perfect sphere. This affinity to smoothing helps the algorithm bypass spurious noise in the image data. Note the volume has increased from the inital four interior lattice points to 1000 interior lattice points by the thirteenth iteration b) shows the effect of the curvature dependent speed on a larger volume. Shown are the first and second iterations. The effects of smoothing are once again very clear.

3.4 Model Simplicity

Simplicity refers to non-intersection, a feature that is shared by all surfaces of real objects. When the model is evolved in time, situations may arise where there is self- intersection, when the model's surface intersects itself. The model then no longer represents a real object, and we must perform a topological transformation to simplify the surface of the model into something more realistic. When provided with a set of interior points and a set of exterior points, a simplicial decomposition of space, as described in section 2.2, provides a means of performing such transformations on a non-simple model locally. The

transformation, in other words, can be done independently on a simplex by simplex basis, without ambiguity.

Intuitively, what the transformation does is it tries to encapsulate disjoint sets of interior points using the reparameterized surface points of the non-simple model. This type of encapsulation can be done on a simplex by simplex basis because given the interior and exterior vertices of a simplex one can always unambiguously construct a hyperplane which separates them. When these hyperplanes are then joined they form simple hypersurfaces which encapsulate points which are on the interior of the model.

The hybrid algorithm uses a simplicial decomposition in handling topological transformations because of the ease of implementation and the fact that it goes hand in hand with the reparameterization algorithm above.

3.5 Extraction of Optimal Model Surface

Since the evolution algorithm used is one of front propagation, the surface will evolve in time, at some iteration i will reach an optimal state where it describes the object in the image data best, and then continue growing until it fills the complete volume block. Now the job of the algorithm becomes determining this iteration i, at which the surface represents the optimal model surface. The hybrid algorithm uses the mean speed of the contour as an indicator of how well the contour fits the object in the image data, and defines i to be the iteration during which the mean displacement of all the points on the surface is smallest. And therefore chooses the surface before the iteration i occurs. Intuitively, it can be seen that this is a reasonable criterion because as a point encounters a high gradient region, usually indicative of an object's surface, it slows down. So you will have a reasonable

accurate model of the object if all points encounter high gradient regions, causing the mean displacement of the points to be smallest.

Chapter 4

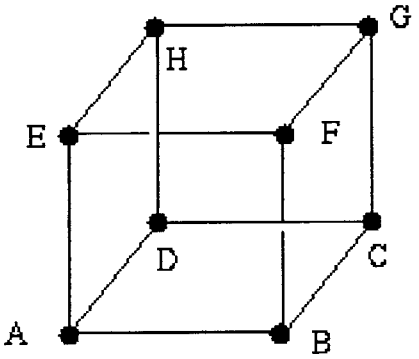
Implementation of the Hybrid Algorithm

4.1 General Implementation Notes

The hybrid algorithm was programmed in C++ using Standard Template Library (STL) data structures, and is written to interface easily with the Visualization Tool Kit (VTK) visualization program. As inputs the algorithm takes a file which contains raw data describing a volume of a predetermined size. As output the program writes a VTK file describing the surface for each iteration of the algorithm, and finally dumps optimal surface to a file called vtkoutopt.vtk.

4.2 Surface Representation

The surfaces are represented as adjacency lists. In essence, each surface is a mapping from a point A to a set of points which represents all the points to which A is connected.



Point	Adjacency List		
A	B	D	E
B	A	C	F
C	B	D	G
D	A	C	H
E	A	F	H
F	B	E	G
G	C	F	H
H	D	E	G

Figure 4.1: This figure shows a cube and its associated adjacency list.

The interface also allows a programmer to output, with the use of the PlaneIterator class, the set of triangular plane segments (represented by three points in $\boldsymbol{R^3}$) that form the surface.

Finally the interface allows the user to individually displace points on the surface while maintaining connectivity.

4.3 Surface Evolution

The evolution of the surface was described in 3.3 as moving each point on the surface of a model in its approximate normal direction. There are two issues which arise here that have to be dealt with. First, the normal at a point on a surface is undefined, since the point represents a discontinuity in the surface. And second, even if the surface were continuous at that point a choice would have to be made between an inward normal and an outward one.

The first of these two issues is resolved by approximating the normal at a point on a surface to be the component based average of the normals of the planes incident to the point. Such that if planes A,B, and C were incident to point p, then the approximated normal at point p would be $(N_A + N_B + N_C)/3$.

The second issue is resolved by traveling one unit in each normal direction and comparing the resulting location with the position of the center of mass of the model. The normal that results in the location furthest from the center of mass of the model is assumed to be the outward normal. In the case where the model has split into more than one component, the center of mass of each individual component must be considered.

In section 3.3, an approximated curvature at a point on a surface was discussed as having an impact on the magnitude of the displacement. This serves as a smoothness constraint. The resulting effect is one in which a point that is in a high approximated curvature region has a larger magnitude of displacement than one in a low approximated curvature region. Again, the approximation is a result of the discontinuity in the surface represented by the point. To approximate the curvature, the algorithm uses a metric which is inversely proportional to the curvature of the area. This metric is simply the sum of the angles made by the planes incident to the point of interest at the point of interest.

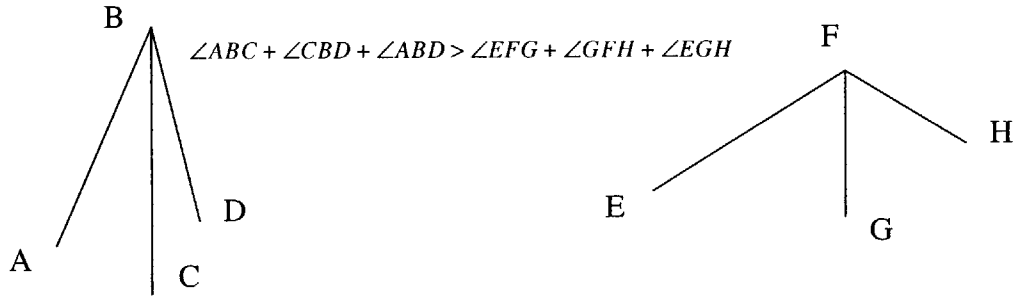


Figure 4.2: This figure shows the relationship between the sum of interior plane angles and the curvature at a point. In this particular example the curvature of B is greater than that of F.

4.4 Simplex Representation

Perhaps the most difficult part of the algorithm was in the representation of the simplicial decomposition of space. The representation of simplex cells was done in such a way as to optimize the reparameterization algorithm. Since the reparameterization algorithm takes each plane in a given surface, and finds all the intersections between the plane and each simplex cell, and since the simplex cells share faces, reparameterization using the

concept of simplex cells would be inefficient since it would find the same reparameterized point many times. Therefore, the simplex cells were represented as a collective as seven sets of parallel lines. Each set with direction either 0,1,2,3,4,5, or 6. Table 4.1 shows the relationship between these direction numbers and the actual direction vectors they represent.

Direction Number	Direction Vector
0	i
1	j
2	k
3	i+j
4	j+k
5	-i+k
6	i+j+k

Table 4.1: This table is used in translating from the internally used direction number to actual direction vectors. Note i = (1,0,0), j = (0,1,0), and k = (0,0,1) are the standard basis direction vectors for R^3

In essence these parallel lines carve out the simplex cells, and so intersections with this grid will correspond with intersections with simplex cells. However, since the algorithm has no notion of what simplex cells are during reparameterization, each reparameterized point is found only once.

Although this simplification makes the reparameterization process relatively easy, the notion of simplex cells cannot be avoided when performing topological transformations.

The topological transformation algorithm is directly expressed in terms of actual simplex cells, and gives restrictions on how the surface model can behave inside a given sim-

plex cell, assuming knowledge of whether or not each of the cell's vertices are inside or outside the surface.

Therefore from simply a reparameterized point and a line with a known direction, the topological transformation algorithm must ascertain which simplex simplex cells have the point on their edge, and be able to access the vertices of those cells to determine for each cell how to restrict the behavior of the surface. The implementation of this algorithm was simplified with the introduction of the SimplexEdge class. The SimplexEdge class takes a line segment and a point, and specifies which simplex edge the point lies on. A simplex edge is simply represented as a location and a direction, and is not bound to any particular cell; instead it is shared by certain cells according to the simplicial grid properties. Table 4.2 shows the table that is used to determine which simplex cells share a certain simplex edge located at (0, 0, 0) for a given direction:

Simplex Edge Direction	Incident Simplex Location	Incident Simplex Type
0	(0, 0, 0)	3
	(0, 0, 0)	4
	(0,-1, 0)	0
	(0, 0,-1)	5
	(0,-1,-1)	1
	(0,-1,-1)	2
1	(0, 0, 0)	0
	(0,0,-1)	1
	(-1, 0, 0)	3
	(-1, 0,-1)	5

Table 4.2: The Table maps SimplexEdges to Simplex Cells which are incident to the edges. A simplex cell consists of a location, which is just the cartesian coordinates of the unit cell that contains the simplex cell, and a type to specify which of the six simplex cells is being referred to. Note: All Simplex Edges in the table start at (0,0,0).

Simplex Edge Direction	Incident Simplex Location	Incident Simplex Type
2	(0, 0, 0)	1
	(0, 0, 0)	4
	(0,-1, 0)	0
	(-1,-1, 0)	2
	(-1,-1, 0)	3
	(-1, 0, 0)	5
3	(0, 0, 0)	0
	(0, 0, 0)	2
	(0, 0, 0)	3
	(0, 0,-1)	1
	(0, 0,-1)	4
	(0, 0,-1)	5
4	(0, 0, 0)	0
	(0, 0, 0)	1
	(0, 0, 0)	2
	(-1, 0, 0)	3
	(-1, 0, 0)	4
	(-1, 0, 0)	5
5	(-1, 0, 0)	4
	(-1, 0, 0)	5
	(-1,-1, 0)	0
	(-1,-1, 0)	2
6	(0, 0, 0)	1
	(0, 0, 0)	2
	(0, 0, 0)	3

Table 4.2: The Table maps SimplexEdges to Simplex Cells which are incident to the edges. A simplex cell consists of a location, which is just the cartesian coordinates of the unit cell that contains the simplex cell, and a type to specify which of the six simplex cells is being referred to. Note: All Simplex Edges in the table start at (0,0,0).

Simplex Edge Direction	Incident Simplex Location	Incident Simplex Type
	(0, 0, 0)	4

Table 4.2: The Table maps SimplexEdges to Simplex Cells which are incident to the edges. A simplex cell consists of a location, which is just the cartesian coordinates of the unit cell that contains the simplex cell, and a type to specify which of the six simplex cells is being referred to. Note: All Simplex Edges in the table start at (0,0,0).

The location of the cells represents the cartesian coordinates of the unit cell in which a given simplex cell is contained. The type of the simplex cell simply represents the location of the simplex cell in the unit cell. Table 4.3 shows the mapping from simplex cell type to simplex vertices for a simplex location of (0, 0, 0):

Simplex Type	Simplex Vertices
0	(0, 0, 0)
	(0, 1, 0)
	(1, 1, 0)
	(0, 1, 1)
1	(0, 0, 0)
	(0, 0, 1)
	(1, 1, 1)
	(0, 1, 1)
2	(0, 0, 0)
	(1, 1, 1)
	(1, 1, 0)
	(0, 0, 1)
3	(0, 0, 0)
	(1, 0, 0)

Table 4.3: This table maps the different simplex cell types into four the four cartesian coordinates that make up the tetrahedral cell. Again this is assuming that the simplex cell is located at (0, 0, 0).

Simplex Type	Simplex Vertices
	(1, 1, 0)
	(1, 1, 1)
4	(0, 0, 0)
	(0, 0, 1)
	(1, 0, 0)
	(1, 1, 1)
5	(1, 0, 0)
	(0, 0, 1)
	(1, 1, 1)
	(1, 0, 1)

Table 4.3: This table maps the different simplex cell types into four the four cartesian coordinates that make up the tetrahedral cell. Again this is assuming that the simplex cell is located at (0, 0, 0).

The coarseness of the reparameterization is in fact controlled by the coarseness of the simplicial grid, or how spaced out the seven sets of parallel lines are. The more spaced out these lines are the less the number of points the new reparameterized surface will contain. Of course when the coarseness is changed each of Tables 4.2 and 4.3 must be adjusted to give the topological transformation algorithms the correct information about the now bigger simplex cells.

4.5 Interesting Optimizations in the Reparameterization Algorithm

Some interesting optimizations were made to the reparameterization algorithm to speed it up. It is easy to see that reparameterization is a huge task. It involves taking each triangular plane segment in an existing surface, and for each plane segment calculating its intersections with seven sets of parallel lines. The bounds checking alone in such a problem is immense. This process is simplified for a given direction n by first projecting the

plane in the -n direction onto an easy to work with plane in R^2 ie. the X-Y plane, the Y-Z plane, or the X-Z plane. So now the lattice points contained within the projected plane will correspond to projections of line segments of the given direction, which intersect the original plane. For a given direction the planes are projected in the following way:

Direction	Direction of Projection	Plane Projected to
0	- i	Y-Z plane
1	- j	X-Z plane
2	- k	X-Y plane
3	- i - j	X-Z plane
4	- j - k	X-Y plane
5	i - k	X-Y plane
6	- i - j - k	X-Y plane

Table 4.4: This table summarizes how a plane is projected by the reparameterization algorithm in order to easily determine which simplex lines of a given direction actually intersect the plane.

Such an optimization allows the algorithm to easily and quickly determine which lines of a given direction actually intersect the plane in question. And reparameterize based solely on those lines.

4.6 Algorithm Summary

This section provides a summary of the algorithm prior to showing the results of testing it. The algorithm has three key phases. It is initialized, iterates, and finally it selects the optimal surface.

The algorithm can be initialized with any surface, anywhere in the surface. Once it is initialized, the algorithm iterates as follows. First it reparameterizes the surface, then it applies a certain displacement to each point on the surface, and finally it performs topological transformations to return the surface to a simple state.

In the final phase the algorithm picks the optimal surface, from all those encountered during the iteration phase.

Chapter 5

Testing

5.1 Testing Methodology

The testing of a segmentation algorithm is an extremely difficult task. This is complicated by the fact that the algorithm presented attempts to segment three dimensional volumes as opposed to two dimensional images. The problem lies in the fact that when one segments real volume or even images, the "correct answer" is usually not known. In other words one cannot compare the result of the algorithm to a correct solution to the segmentation problem because this correct solution is an unknown.

As a result the algorithm presented was tested only with synthetic data so that the results could be compared to the known solution to the problem.

Another problem encountered with a three dimensional segmentation algorithm is the amount of memory it consumes. The algorithm presented in this thesis requires on the order of n^2 bytes where n is the total number of voxels in the volume. This is because at each iteration the algorithm holds an adjacency list containing the surface from the last iteration. An adjacency list of a graph contains a list of points, and for each point, a list of points that point is connected to. So one can imagine, in a fully connected graph, one would require enough memory to hold n^2 points. The maximum number of points is simply the number of point in the volume, and if each point is represented by a single byte, we can easily see that the algorithm requires n^2 bytes of memory.

So for example if the volume were 100x100x100, or contained 1 million voxels, the algorithm would require on the order of 10^{12} bytes. Although this is a worst case scenario

and the multiplicative constant is small, this memory constraint is still imposing when attempting to segment large volumes, as seen with real volumes.

5.2 The Data Set

The results presented in this thesis are all results obtained from segmenting synthetic binary volumes. However, special care was taken to ensure that the surface would be irregular, and would at least resemble real data in shape. The slices were produced using a drawing program, and inputted into the program as arrays of raw data.

The set of slices shown in Figure 5.1 were used to create the volume. Each slice is 30x30 pixels, and there are 30 slices.

Figure 5.1: This is the set of slices used to create the volume. The shapes were chosen so as to be irregular, yet identifiable in the output. The volume was composed by using purely white images for slices 1-6 and 24-30, the first image in the set for slices 7-11, the second image in the set for slices 12-18, and finally the last image for slices 19-23.

Chapter 6

Results

6.1 Volume Preprocessing

As a preprocessing step, necessary to obtain the image dependent speed term, the algorithm first computes the magnitude of the three dimensional gradient. Figure 6.1 shows the results of this preprocessing steps. As one can see the image gradient magnitude is affected by neighbouring slices.

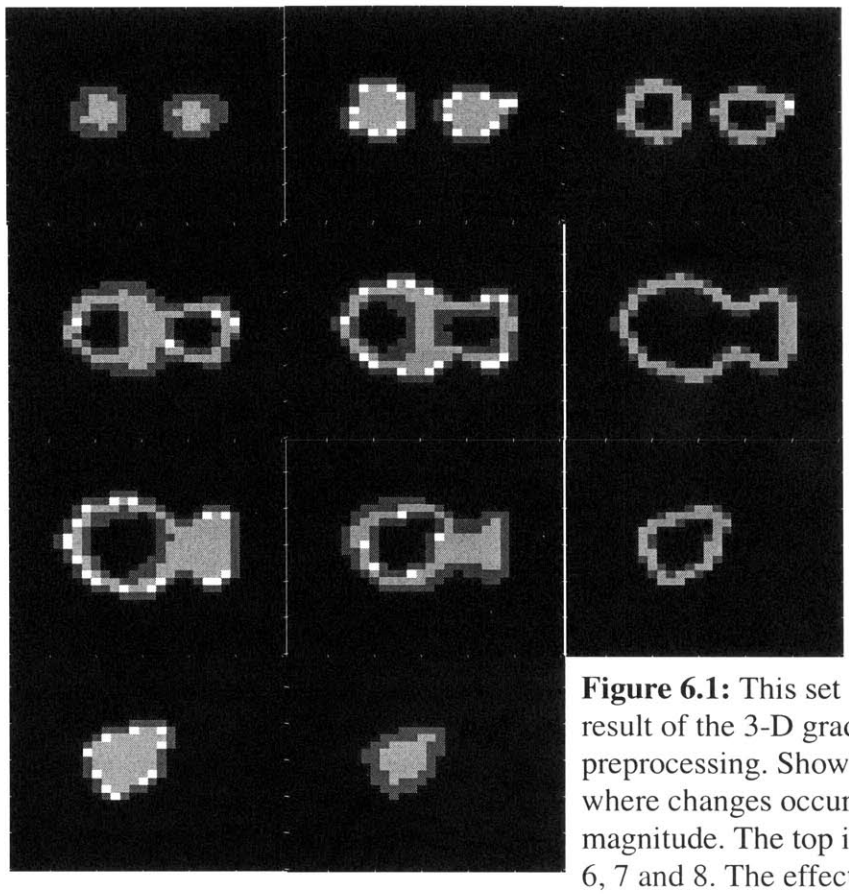


Figure 6.1: This set of images is the result of the 3-D gradient magnitude preprocessing. Shown are the slices where changes occur in the gradient magnitude. The top images are slices 6, 7 and 8. The effects of the gradient being taken in three dimensions can already be seen: from figure 5.1 one can see that the volume starts at slice 7, however one can see that slice 6 of the image gradient magnitude has some non-zero intensities. The second set of images shows slices 11, 12 and 13. Here again we see the effects of neighbouring slices on the magnitude of the volume gradient. The third and fourth sets are slices 18, 19, 20 and slices 23 and 24 respectively.

The algorithm can now use these slices to determine the speed of any point in space. It does this by scaling the intensities of the integer lattice points surrounding it by the distance the point is from them, and using the recoprical of the result as the speed of the point.

6.2 Surface Evolution

The surface evolved as expected filling the object in the volume up, and then smoothing the surface. Two views of the results will be shown: a top-down view, which is the result of looking at the surface from the positive z-axis, and a side view, which is the result of looking at the surface from the negative x-axis.

Figure 6.2 shows the initial surface used in the algorithm. Notice how this surface in no way resembles the real surface of the object being segmented.

Figure 6.2: The set of images shows the surface intialization.

Figure 6.3 shows the surface after the first iteration. We can see the curvature dependent speed in play here, as the surface becomes more spherical in nature, yet the surface has not yet hit any gradients and so expands freely without any image constraints.

Figure 6.3: The set of images shows the surface after the first iteration of the algorithm.

Figure 6.4 shows the surface after 3 iterations of the algorithm, and here we see that the algorithm has started to become constrained by image gradients present.

Figure 6.4: The set of images shows the surface after three iterations. The surface has flattened out in the top-down view, and is taking the form of the middle slice in figure 5.1.

Figure 6.5 shows the surface after 7 iterations of the algorithm, and we can now more clearly see the effects of the volume gradients, and the features of the input slices.

Figure 6.5: The set of images shows the surface after seven iterations.

Figure 6.6 shows the surface after 12 iterations, one can now easily identify the features of the input slices, including the fish like shape of the second set of slices in the top-down view, and the large bump of the last set of slices and two small bumps of the first set

of slices in the side view. However, one can also see the surface is pretty jagged, and not very smoothed out. This is mainly because the surface has just started hitting the large image gradients.

Figure 6.6: The surface after 12 iterations of the algorithm.

Figure 6.7 shows the surface now after 20 iterations, the features are now much clearer to see. We can also observe that changes in the surface are occuring more slowly due to the high image gradients encountered by the surface. The surface also appears smoother than the previous due to the curvature dependent speed term.

Figure 6.7: The surface after 20 iterations of the algorithm.

Figure 6.8 shows the surface after 40 iterations, we can still see the features as the algorithm is moving through the high gradient borders very slowly, but now the surface is much smoother again due to the curvature dependent speed term.

Figure 6.8: The surface after 40 iterations of the algorithm.

Figure 6.9 shows the surface after 67 iterations, the surface now has finally moved passed the high gradient regions of the volume and is now starting to leak out causing inaccuracies in the surface. The algorithm has passed through the boundaries of the object in the volume.

Figure 6.9: The surface after 67 iterations of the algorithm. The surface has leaked out and is no longer a good representation of the surface.

6.3 Optimal Surface Identification

The algorithm also recommends one of the surfaces as being the optimal description of the object in the volume. As discussed in chapter 4, the optimal surface is chosen by the algorithm as the surface which generates the smallest average displacement over all points on the surface. For this particular data set the algorithm picked the surface at iteration 38.

Figure 6.10 shows the top-view and side-view images of this surface.

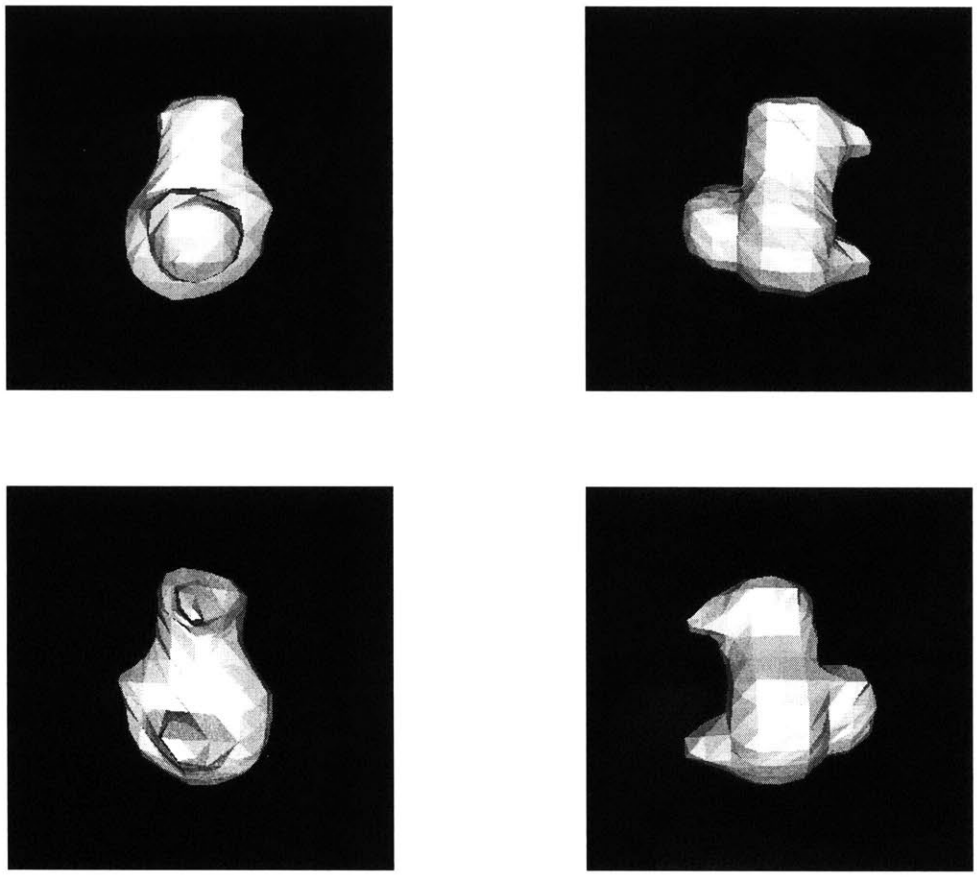


Figure 6.10: This set of surfaces, shows the optimal surface. The first image shows the big bump found in the third set of slices in the volume, and also the shape of the second set of slices. The second image shows a side view of the surface in which both the little humps found in the first set of slices and also the big hump found in the third set of slices are clearly visible. The third image gives a bottom-up view of the surface, and finally the fourth message shows the surface from the other side view (ie. from the positive x-axis).

6.4 Evaluation of Results

Although the algorithm was only tested on synthetic binary volumes, the major features of the algorithm were very clearly exhibited. The algorithm did a good job extracting the geometry of the object from the volume, but it also showed its ability to perform topological transformations, its ability to reparameterize the surface at each iteration, its ability

to move the surface with curvature dependent speed, and its ability to find the high gradient regions of the image. Most of these features were already shown in chapter 3, while discussing the design of the algorithm. However, two features, the topological transformation and the ability to transition between coarseness levels, were not visually exhibited, but there effects become clear when we view the results of the segmentation.

The topological transformations keep the surface simple at every iteration. It is easy to take this function for granted, however when we think about the possibility of two points crossing during an iteration, we can easily see the use of such a function.

The coarseness change during the running of the algorithm is also an interesting feature of the algorithm. It lessens the amount of memory required to run the algorithm when the surface increases in size. The actual implementation changes to a coarseness factor of 2 after the tenth iteration of the algorithm.

The accuracy of the results was quite impressive. The algorithm was able to segment volume features which were five pixels in depth, and clearly show them protruding the surface. Although the algorithm was not tested in the presence of noise, some things can be deduced from its performance in a noise-free environment. The algorithm took about 15 iterations to get a pretty close approximation to the surface of the object, maintained that surface for about 45 iterations and then leaked out of the object for the last 10 or so iterations. The 45 iteration buffer where the surface moves very little is why I believe the algorithm would perform well if tested in the presence of noise. In these 45 iterations, the algorithm should be able to overcome (using the curvature dependent speed term) any noise within the boundaries of the object.

In terms of actual running time of the program, for the 30x30x30 volume, switching to a coarseness factor of 2 after the 10th iteration, the program took about 3.5 hours to run on a Sun ULTRA 10 at Athena.

Chapter 7

Conclusion

7.1 Drawbacks and Shortcomings of Algorithm and Research

One major drawback of the algorithm is the fact that it has not yet been tested on real data. Because of the adjacency lists maintained throughout the iterations, the algorithm works accurately only on small data sets. The algorithm can work on large data sets but this would require making the coarsenss factor large in order to minimize the amount of memory needed, in turn this would reduce the accuracy of the algorithm. Also the algorithm has not yet been tested on data with noise, which is common in real data. However, because of the solidity of the mechanics of the algorithm, it is almost certain that the algorithm would perform as well on noisy data.

The image dependent speed term might also be a parameter that one might want to play with. Currently, the algorithm has been programmed to use an inverse gradient magnitude speed term. And although it seems to work well, one might also want to try using an inverse exponential speed term, which might be more effective on noisier data.

Finally, the way in which the outward normal is chosen assumes a relatively regular shape. The outward normal is assumed to be the normal which if followed will lead a point further away from the center of mass of the current surface. However, one can imagine situations and irregular shapes which would give such a scheme difficulty.

7.2 Recommended Future Work

Perhaps the most important extension to this research that must be done in order for the algorithm to be able to segment real data sets, is the more efficient usage of memory in

the surface representation, and topological transformation data structures. This will enable the algorithm to be able to segment real data which is usually 256x256x128 two byte words, or 16 megabytes. After such extensions have been made, one should improve the versatility of the algorithm by discovering better heuristics for identifying the outward normal to a surface. One should also investigate the use of different speed terms on the result of the algorithm. The framework exists in this thesis to segment almost anything. All that is required is slight modifications to the algorithm.

7.3 In Conclusion

In conclusion, the thesis presented an accurate, robust, and versatile hybrid three dimensional segmentation algorithm, which has the simplicity of the marching cubes algorithm and the power of sophisticated algorithms such as front propagation using level sets or topologically adaptable surfaces. With the implementation of the extensions suggested above, the algorithm should provide an easy to use, easy to understand and versatile solution to the problem of three dimensional segmentation.

Appendix A

3D Geometry Library and Source Code

A.1 Header File (3dgeometry.h)

```
#include <stdlib.h>
#include <iostream.h>
#include <list.h>

#ifndef a_h
#define a_h


/*=============================================================================*/
/* The compare class provides a less than operator for points */
/* that the STL needs in order to keep the map in order      */



class IsPointException{ };

class NoIntersectionException{ };

class point{
 //Three dimensional representation of a point.
 //a point corresponds to an ordered triple (x,y,z) of floats
 //it represents a location in three dimensional space.

 float x,y,z;

 //private methods: setX(), setY(), and setZ()
 void setX(float);
 void setY(float);
 void setZ(float);


public:

 //constructors: initiates with three floats x,y,z
 //            default constructor
 //            and copy constructor

 point(float,float,float);
 point();
 point(const point &);
 //methods

 //assignment operator and equality check.
 //Also we must provide a system of strict ordering for points
 //this is done by first comparing z, then y, then x.
```

```
  void operator=(const point&);
  bool operator==(const point&) const;
  bool operator>(const point&) const;
  bool operator<(const point &) const;
  bool operator>=(const point &) const;
  bool operator<=(const point &) const;
  point operator-(const point &) const;
  point operator+(const point &) const;
  point operator*(const float &)const;
  point operator/(const float&)const;

  //simple ways to get values of points
   float getX() const;
   float getY() const;
   float getZ() const;
   point closestPoint();


};


//General Vector functions, using the point class.
//Dot Product, Cross Product, Vector Magnitude, and Angle Between Vectors


float magnitude(const point&);

float dot(const point &, const point &);

point cross(const point &,const point &);

float angleBetweenVectors(const point &, const point &);

point normalize(const point&);

float distanceBetweenPoints(const point&,const point&);


class line_segment{
  //This class represents a line segment in 3-Dimensions.
  //It consists of a start 3D point and an end 3D point.
  //p1 and p2 are unordered ie. line has no direction.
  //p1 and p2 must be distinct.

  point p1;
  point p2;

 protected:

  void setp1(point);
  void setp2(point);


public:
```

```
 //constructors: construct given two points a start and an end
 //            copy constructor

 line_segment(point,point) throw(IsPointException);
 line_segment(const line_segment&);
 line_segment();

 //Application Specific Constructor

 line_segment(int /*direction*/,point);
 void getGridCrossings(list<point> &, int /*grid courseness*/);


 //assignment operator

 void operator=(const line_segment &);
 bool operator==(const line_segment &) const;

 //line_segment utilities

 point intersects(line_segment) throw(NoIntersectionException);

 point getp1() const;

 point getp2() const;

 bool isPointOn(point);

};


class plane_segment{
 //This class represents a segment of a plane.
 //It is represented as three points p1,p2,p3 the largest number of points
 //that still gauranties coplaniarity.
 //p1,p2 and p3 are unordered and must be distinct.

 point p1;
 point p2;
 point p3;
 point Normal1;
 point Normal2;

 void setpl1(point);
 void setpl2(point);
 void setpl3(point);
 void setNormal1(point);
 void setNormal2(point);


public:

 //constructors: Constructor taking three distinct points.
 //            copy constructor.
```

```
  plane_segment(point,point,point);
  plane_segment(const plane_segment &);

  //operators: assignment and equivalence

  void operator=(const plane_segment &);
  bool operator==(const plane_segment &) const;

  //useful operators.

  line_segment intersectsPlane(plane_segment) throw(NoIntersectionException);
  point intersectsLine(line_segment);
  bool pointOnPlane(point);
  point getpl1() const;
  point getpl2() const;
  point getpl3() const;
  point getNormal1() const;
  point getNormal2() const;

  //Application Specific Functions

  void whichSimplexLinesIntersect(list<line_segment> &,int /*direction*/, int /*grid coarseness*/);
  plane_segment project(int /*direction*/);
  void whichPointsAreIn(list<point> &, int /*grid coarseness*/);

};



ostream& operator<<(ostream& s,point &p);

ostream& operator<<(ostream& s,line_segment &l);

ostream& operator<<(ostream& s,plane_segment &p);



class comparepoints{
public:
  bool operator()(point,point);
};


#endif
```

A.2 Source Code (3dgeometry.cpp)

```
/*** 3-D Geometry Library... To be used with Segmentation Algorithm****/
/*** Author: Varouj A. Chitilian****/
/*** Date: Nov 10, 1998***/
/*** Email: vchitil@mit.edu,vchitilian@sarnoff.com***/
```

```
#include <stdlib.h>
#include <iostream.h>
#include <list.h>
#include <math.h>



//These are exceptions that will be thrown, when the occasion arises

//Is point Exception will ensure and will not allow line segments to be defined
//if the start and end points are the same.
class IsPointException{};

//No intersection Exception will inform user when the geometric entities do not
//intersect.

class NoIntersectionException{};



//The class point is the most basic geometric entity in 3 dimensions.

class point{
 //Three dimensional representation of a point.
 //a point corresponds to an ordered triple (x,y,z) of floats
 //it represents a location in three dimensional space.

 float x,y,z;

 //private methods: setX(), setY(), and setZ()
 void setX(float);
 void setY(float);
 void setZ(float);


public:

 //constructors: initiates with three floats x,y,z
 //            default constructor
 //            and copy constructor

 point(float,float,float);
 point();
 point(const point &);
 //methods

 //assignment operator and equality check.
 //Also we must provide a system of strict ordering for points
 //this is done by first comparing z, then y, then x.
```

```
 void operator=(const point&);
 bool operator==(const point&) const;
 bool operator>(const point&) const;
 bool operator<(const point &) const;
 bool operator>=(const point &) const;
 bool operator<=(const point &) const;
 point operator-(const point &) const;
 point operator+(const point &) const;
 point operator*(const float &)const;
 point operator/(const float &) const;

 //simple ways to get values of points
 float getX() const;
 float getY() const;
 float getZ() const;
 point closestPoint();

};


//only for points with positive coordinates

point point::closestPoint(){

 float x = getX();
 float y = getY();
 float z = getZ();


 int ix = (int) x;
 int iy = (int) y;
 int iz = (int) z;

 if (x>ix+0.5){

  if (y>iy+0.5){

   if (z>iz+0.5)
    return point(ix+1,iy+1,iz+1);
   else
    return point(ix+1,iy+1,iz);
  }
  else{

   if (z>iz+0.5)
    return point(ix+1,iy,iz+1);
   else
    return point(ix+1,iy,iz);
  }
 }
 else{
  if (y>iy+0.5){

   if (z>iz+0.5)
```

```
      return point(ix,iy+1,iz+1);
    else
      return point(ix,iy+1,iz);
   }
   else{

    if (z>iz+0.5)
      return point(ix,iy,iz+1);
    else
      return point(ix,iy,iz);
   }
  }

}



void point::setX(float x0){
 x = x0;
}

void point::setY(float y0){
 y = y0;
}

void point::setZ(float z0){
 z =z0;
}

float point::getX() const{
 return x;
}

float point::getY() const{
 return y;
}

 float point::getZ() const{
 return z;
}

point::point(float x0,float y0,float z0){
 setX(x0);
 setY(y0);
 setZ(z0);
}

point::point(){
 setX(0);
 setY(0);
 setZ(0);
}

point::point(const point &p){
```

```
 setX(p.x);
 setY(p.y);
 setZ(p.z);
}


void point::operator=(const point & p){
 setX(p.getX());
 setY(p.getY());
 setZ(p.getZ());
}

bool point::operator==(const point & p) const{
 if(p.getX() == getX() &&
    p.getY() == getY() &&
    p.getZ() == getZ())
   return 1;
 else return 0;
}



bool point::operator<(const point &p) const{
 if((getZ()<p.getZ())||
    (getZ()==p.getZ() && getY() < p.getY()) ||
    (getZ() ==p.getZ() && getY() ==p.getY() && getX()<p.getX()))
   return 1;
 return 0;
}

bool point::operator>(const point&p) const{
 if(((*this)<p) || (*this)==p)
   return 0;
 return 1;
}


bool point::operator>=(const point &p) const{
 if ((*this)<p)
   return 0;
 return 1;
}

bool point::operator<=(const point &p) const{
 if ((*this)>p)
   return 0;
 return 1;
}


point point::operator-(const point&p) const{
 return point(getX()-p.getX(),getY()-p.getY(),getZ()-p.getZ());
}
```

```
point point::operator+(const point & p) const{
 return point(getX()+p.getX(),getY()+p.getY(),getZ()+p.getZ());
}


point point::operator*(const float & i)const{
 return point(getX()*i,getY()*i,getZ()*i);
}


point point::operator/(const float &f) const{
 return point(getX()/f,getY()/f,getZ()/f);
}



 //Output stream formation functions
 //points
ostream& operator<<(ostream& s,point &p){
 //Stream output operator for points.

 return s<<" ("<<p.getX()<<","<<p.getY()<<","<<p.getZ()<<") ";
}
//General Vector functions, using the point class.
//Dot Product, Cross Product, Vector Magnitude, and Angle Between Vectors
//Normalization

float magnitude(const point&);

float dot(const point &, const point &);

point cross(const point &,const point &);

float angleBetweenVectors(const point &, const point &);

point normalize(const point &);

float distanceBetweenPoints(const point &,const point &);

float magnitude(const point & p){
 return sqrt(p.getX()*p.getX() + p.getY()*p.getY() + p.getZ()*p.getZ());
}

float dot(const point &a, const point &b){
 return (a.getX()*b.getX() + a.getY()*b.getY() + a.getZ()*b.getZ());
}

point cross(const point &a,const point &b){
  float a1 = a.getX();
  float a2 = a.getY();
  float a3 = a.getZ();

  float b1 = b.getX();
  float b2 = b.getY();
```

```
 float b3 = b.getZ();

 return (point(a2*b3 - a3*b2,a3*b1 -a1*b3,a1*b2 - a2*b1));
}

float angleBetweenVectors(const point &a, const point &b){

 float angle;
 float result;

 result = (dot(a,b)/(magnitude(a) * magnitude(b)));

 if (result <-1){
  // cout<<"we got problems result is less than -1 cannot take acos... exiting";
  //exit(0);
  result = -1;
 }
 if (result >1.01){
  cout<<"\nA = "<<a;
  cout<<"\nB = "<<b;
  cout<<"\na dot b = "<<dot(a,b)<<"\n";
  cout<<"magnitude a = "<<magnitude(a)<<"\n";
  cout<<"magnitude b = "<<magnitude(b)<<"\n";

  cout<< result<<"\n";

  //cout<<"we got problems result is greater than 1 can't take acos... exiting";
  //exit(0);
 result =1;
 }

 angle = acos(result);

 return angle/6.3;
 //scaling angle so that it is between 0 and 0.5

}

float distanceBetweenPoints(const point&a,const point&b){

 float x = a.getX() - b.getX();
 float y = a.getY() - b.getY();
 float z = a.getZ() - b.getZ();

 return (x*x + y*y + z*z);
}

point normalize(const point & p){

 return (p/magnitude(p));
}

class line_segment{
 //This class represents a line segment in 3-Dimensions.
```

```
 //It consists of a start 3D point and an end 3D point.
 //p1 and p2 are unordered ie. line has no direction.
 //p1 and p2 must be distinct.

 point p1;
 point p2;

protected:
 void setp1(point);
 void setp2(point);


public:
 //constructors: construct given two points a start and an end
 //            copy constructor

 line_segment(point,point) throw(IsPointException);
 line_segment(const line_segment&);
 line_segment();

 //Application Specific Constructor

 line_segment(int /*direction*/,point);
 void getGridCrossings(list<point> &, int/*grid coarseness*/);



 //assignment operator

 void operator=(const line_segment &);
 bool operator==(const line_segment &) const;


 //line_segment utilities

 point intersects(line_segment) throw(NoIntersectionException);

 point getp1() const;

 point getp2() const;

 bool isPointOn(point);

};

void line_segment::setp1(point a){
 p1 =a;
}

void line_segment::setp2(point b){
 p2=b;
}

point line_segment::getp1() const{
```

```
 return p1;
}

point line_segment::getp2() const{
 return p2;
}


line_segment::line_segment(point a,point b) throw (IsPointException){
 if (a==b) throw IsPointException();
 setp1(a);
 setp2(b);
}

line_segment::line_segment(){
}

line_segment::line_segment(const line_segment & l){
 setp1(l.getp1());
 setp2(l.getp2());
}



//Application Specific Constructor

line_segment::line_segment(int direction,point p){


 switch (direction){

 case 0:
  setp1(point(0,p.getX(),p.getY()));
  setp2(point(1024,p.getX(),p.getY()));
  break;

 case 1:
  setp1(point(p.getX(),0,p.getY()));
  setp2(point(p.getX(),1024,p.getY()));
  break;

 case 2:
  setp1(point(p.getX(),p.getY(),0));
  setp2(point(p.getX(),p.getY(),1024));
  break;

 case 3:
  setp1(point(p.getX(),0,p.getY()));
  setp2(point(p.getX()+1024,1024,p.getY()));
  break;

 case 4:
  setp1(point(p.getX(),p.getY(),0));
  setp2(point(p.getX(),p.getY()+1024,1024));
```

```
  break;

 case 5:
  setp1(point(p.getX(),p.getY(),0));
  setp2(point(p.getX()-1024,p.getY(),1024));
  break;

 case 6:
  setp1(point(p.getX(),p.getY(),0));
  setp2(point(p.getX()+1024,p.getY()+1024,1024));
  break;
 }
}
//Application Specific function Should only be used with line_segments that are 2-D
//Intended for 2 space geometry only.

void line_segment::getGridCrossings(list<point> & lpoints, int coarse){

 int miny,maxy;
 float y1,y2;
 y1 = getp1().getY();
 y2 = getp2().getY();



 if(y1> y2){


  if(y2>0){
   int i3;
   for(i3 = (int) y2+1; i3%coarse!= 0;i3++){}
   miny = i3;
   int i4;
   for(i4 = (int) y1;i4%coarse != 0;i4--){}
   maxy = i4;
  }
  if(y2<0 && y1>0){
   int j3;
   for(j3 =(int) y2;j3%coarse!=0;j3++){}
   miny = j3;
   int j4;
   for(j4 = (int) y1;j4%coarse !=0;j4--){}
   maxy = j4;
  }
  if(y2<0 && y1<0){
   int k3;
   for(k3 =(int) y2;k3%coarse!=0;k3++){}
   miny = k3;
   int k4;
   for(k4 = (int) y1-1;k4%coarse!=0;k4--){}
   maxy = k4;
  }
 }
 else{
```

```
  if(y1>0){
    int i5;
    for(i5 = (int) y1+1;i5%coarse !=0;i5++){}
    miny = i5;
    int i6;
    for(i6 = (int) y2;i6%coarse !=0;i6--){}
    maxy = i6;
  }
  if(y1<0 && y2>0){
    int j5;
    for(j5 =(int) y1;j5%coarse!=0;j5++){}
    miny = j5;
    int j6;
    for(j6 = (int) y2;j6%coarse !=0;j6--){}
    maxy = j6;
  }

  if(y1<0 && y2<0){
    int k5;
    for(k5 =(int) y1;k5%coarse!=0;k5++){}
    miny = k5;
    int k6;
    for(k6 = (int) y2-1;k6%coarse!=0;k6--){}
    maxy = k6;
  }
 }


 float m = (getp2().getY()-getp1().getY())/(getp2().getX() -getp1().getX());

 float adjust = getp1().getX() - (1/m)*(getp1().getY());

 for (int ycoor = miny;ycoor <maxy +1;ycoor= ycoor+coarse){
  lpoints.push_back(point ((1/m)*ycoor + adjust,ycoor,0));
 }
}



//operators

void line_segment::operator=(const line_segment &l){
 setp1(l.getp1());
 setp2(l.getp2());
}

bool line_segment::operator==(const line_segment &l) const{
 if(getp1()==l.getp1() && getp2()==l.getp2())
  return 1;
 return 0;
}

//utilities
```

```
/******** PLANE SEGMENT CLASS **********/

class plane_segment{
 //This class represents a segment of a plane.
 //It is represented as three points p1,p2,p3 the largest number of points
 //that still gauranties coplaniarity.
 //p1,p2 and p3 are unordered and must be distinct.

 point p1;
 point p2;
 point p3;

 //Each plane has two normals...
 //both will be stored when plane is instantiated, and
 //either Normal1 or Normal2 will be picked at run time
 //by the algorithm, depending on which one is pointing outward
 //from the surface.

 point Normal1;
 point Normal2;

 void setpl1(point);
 void setpl2(point);
 void setpl3(point);
 void setNormal1(point);
 void setNormal2(point);

public:

 //constructors: Constructor taking three distinct points.
 //             copy constructor.

 plane_segment(point,point,point);
 plane_segment(const plane_segment &);

 //operators: assignment and equivalence

 void operator=(const plane_segment &);
 int operator==(const plane_segment &) const;

 //useful operators.

 line_segment intersectsPlane(plane_segment) throw(NoIntersectionException);
 point intersectsLine(line_segment);
 bool pointOnPlane(point);
 point getpl1() const;
 point getpl2() const;
 point getpl3() const;
 point getNormal1() const;
 point getNormal2() const;
```

```
  //Application Specific Functions

  void whichSimplexLinesIntersect(list<line_segment> &,int /*direction*/, int /*grid coarseness*/);
  plane_segment project(int /*direction*/);
  void whichPointsAreIn(list<point> &, int /*grid coarseness*/);

};


//constructors


void plane_segment::setpl1(point a){
  p1 =a;
}

void plane_segment::setpl2(point b){
  p2 =b;
}

void plane_segment::setpl3(point c){
  p3 =c;
}

void plane_segment::setNormal1(point c){
  Normal1 = c;
}

void plane_segment::setNormal2(point c){
  Normal2 = c;
}

point plane_segment::getpl1() const{
  return p1;
}

point plane_segment::getpl2() const{
  return p2;
}

point plane_segment::getpl3() const{
  return p3;
}

point plane_segment::getNormal1() const{
  return Normal1;
}

point plane_segment::getNormal2() const{
  return Normal2;
}


plane_segment::plane_segment(point a,point b,point c){
```

```
 setpl1(a);
 setpl2(b);
 setpl3(c);

 //d and e are vectors

 point d(b-a);
 point e(c-a);

 //f is the cross product of d and e vectors

 point f(cross(d,e));

 point g(-1*(f.getX()),-1*(f.getY()),-1*(f.getZ()));

 setNormal1(f);
 setNormal2(g);

}


plane_segment::plane_segment(const plane_segment & p){
 setpl1(p.getpl1());
 setpl2(p.getpl2());
 setpl3(p.getpl3());
 setNormal1(p.getNormal1());
 setNormal2(p.getNormal2());
}

void plane_segment::operator=(const plane_segment &p){
 setpl1(p.getpl1());
 setpl2(p.getpl2());
 setpl3(p.getpl3());
 setNormal1(p.getNormal1());
 setNormal2(p.getNormal2());
}



point plane_segment::intersectsLine(line_segment l){

 point returnpoint;
 point pdiff(l.getp2()-l.getp1());
 int max;

 if(pdiff.getX()>pdiff.getY() && pdiff.getX()>pdiff.getZ())
   max = (int) pdiff.getX();
 else
   if(pdiff.getY()>pdiff.getZ())
   max = (int) pdiff.getY();
   else
     max = (int) pdiff.getZ();
```

```
point ptest(point (pdiff.getX()/max,pdiff.getY()/max, pdiff.getZ()/max));

point p1(getpl1());
point p2(getpl2());
point p3(getpl3());

point p4(l.getp1());
point p5(l.getp2());

point vector1(p2-p1);
point vector2(p3-p1);

point normal(vector1.getY()*vector2.getZ() - vector1.getZ()*vector2.getY(),
        vector1.getZ()*vector2.getX() - vector1.getX()*vector2.getZ(),
        vector1.getX()*vector2.getY() - vector1.getY()*vector2.getX());

float planeconst = normal.getX()*p1.getX() +  normal.getY()*p1.getY() +
 normal.getZ()*p1.getZ();

// equation of plane is represented by
// normal.x* X + normal.y* Y + normal.z* Z = planeconst

if(ptest == point(1,0,0)){
 //this means that y and z will be the same as in the line.

 returnpoint = point((planeconst-normal.getY()*p4.getY()-
     normal.getZ()*p4.getZ())/normal.getX(),
    p4.getY(),p4.getZ());
}
if(ptest == point(0,1,0)){
 //This means that x and z will be the same as in the line.

 returnpoint = point(p4.getX(),(planeconst-normal.getX()*p4.getX()-
      normal.getZ()*p4.getZ())/normal.getY(),
    p4.getZ());
}

if(ptest == point(0,0,1)){
 //this means that x and y will be the same as in the line.

 returnpoint = point(p4.getX(),p4.getY(),
    (planeconst-normal.getY()*p4.getY()-
     normal.getX()*p4.getX())/normal.getZ());
}

if(ptest == point (1,1,0)){

 int K = (int) p4.getX() - (int) p4.getY();

 returnpoint = point(K+((planeconst-normal.getZ()*p4.getZ()-normal.getX()*K)/
       (normal.getX() +normal.getY())),
    ((planeconst-normal.getZ()*p4.getZ()-normal.getX()*K)/
       (normal.getX() +normal.getY())),
    p4.getZ());
```

```
 }

 if (ptest == point (0,1,1)){

  int K = (int) p4.getZ() - (int)p4.getY();

  returnpoint = point(p4.getX(),
     ((planeconst-normal.getX()*p4.getX()-normal.getZ()*K)/
      (normal.getY()+normal.getZ())),
     ((planeconst-normal.getX()*p4.getX()-normal.getZ()*K)/
      (normal.getY()+normal.getZ()))+K);
 }

 if (ptest == point (-1,0,1)){

  int K = (int) p4.getX() + (int) p4.getZ();

  returnpoint = point(((planeconst - normal.getY()*p4.getY() - normal.getZ()*K)/
      (normal.getX()-normal.getZ())),
     p4.getY(),
     K - ((planeconst -normal.getY()*p4.getY() - normal.getZ()*K)/
        (normal.getX()-normal.getZ())));

 }

 if (ptest == point (1,1,1)){

  returnpoint = point(((planeconst - normal.getX()*p4.getX() - normal.getY()*p4.getY()
       -normal.getZ()*p4.getZ())/
      (normal.getX()+normal.getY()+normal.getZ())) + p4.getX(),
     ((planeconst - normal.getX()*p4.getX() - normal.getY()*p4.getY()
       -normal.getZ()*p4.getZ())/
      (normal.getX()+normal.getY()+normal.getZ())) + p4.getY(),
     ((planeconst - normal.getX()*p4.getX() - normal.getY()*p4.getY()
       -normal.getZ()*p4.getZ())/
      (normal.getX()+normal.getY()+normal.getZ())) + p4.getZ());
 }

 return returnpoint;
}



//Application Specific functions

plane_segment plane_segment::project(int direction){
 point newp1;
 point newp2;
 point newp3;

 switch (direction){

 case 0:
```

```
  newp1 = point(getpl1().getY(),getpl1().getZ(),0);
  newp2 = point(getpl2().getY(),getpl2().getZ(),0);
  newp3 = point(getpl3().getY(),getpl3().getZ(),0);
  break;

case 1:
  newp1 = point (getpl1().getX(),getpl1().getZ(),0);
  newp2 = point(getpl2().getX(),getpl2().getZ(),0);
  newp3 = point (getpl3().getX(),getpl3().getZ(),0);
  break;

case 2:
  newp1= point (getpl1().getX(),getpl1().getY(),0);
  newp2= point (getpl2().getX(),getpl2().getY(),0);
  newp3= point (getpl3().getX(),getpl3().getY(),0);
  break;

case 3:
   newp1= point (getpl1().getX()-getpl1().getY(),
     getpl1().getZ(),0);
   newp2= point (getpl2().getX()-getpl2().getY(),
     getpl2().getZ(),0);
   newp3= point (getpl3().getX()-getpl3().getY(),
     getpl3().getZ(),0);
  break;

case 4:
   newp1= point (getpl1().getX(),
     getpl1().getY()-getpl1().getZ(),0);
   newp2= point (getpl2().getX(),
     getpl2().getY()-getpl2().getZ(),0);
   newp3= point (getpl3().getX(),
     getpl3().getY()-getpl3().getZ(),0);
   break;

case 5:
   newp1= point (getpl1().getX()+getpl1().getZ(),
     getpl1().getY(),0);
   newp2= point (getpl2().getX()+getpl2().getZ(),
     getpl2().getY(),0);
   newp3= point (getpl3().getX()+getpl3().getZ(),
     getpl3().getY(),0);
   break;

case 6:
   newp1= point (getpl1().getX()-getpl1().getZ(),
     getpl1().getY()-getpl1().getZ(),0);
   newp2= point (getpl2().getX()-getpl2().getZ(),
     getpl2().getY()-getpl2().getZ(),0);
   newp3= point (getpl3().getX()-getpl3().getZ(),
     getpl3().getY()-getpl3().getZ(),0);
  break;

}
```

```
 return plane_segment(newp1,newp2,newp3);

}



//line segments

ostream& operator<<(ostream& s,line_segment &l){
 //Stream output operator for line segments.

 return s<<"line = [p1: "<< l.getp1() <<",p2 :"<<l.getp2()<<"]";
}


//plane segments

ostream& operator<<(ostream& s,plane_segment &p){
 //Stream output operator for plane segments.

 return s<<"plane = [p1: "<< p.getpl1() <<",p2: "<<p.getpl2()<<",p3: "<<p.getpl3()<<"]";
}




//Debug this function so that it is negative and zero friendly
//This function should only be used by planes lying on the x-y plane
//in other words planes that use this function should be 2-D

void plane_segment::whichPointsAreIn(list<point> &returnpoints, int coarse){

 line_segment larray[3];

 larray[0] = line_segment(getpl1(),getpl2());
 larray[1] = line_segment(getpl1(),getpl3());
 larray[2] = line_segment(getpl2(),getpl3());

 float minyf,maxyf;
 int miny,maxy;

 //Establishing range of plane

 minyf = getpl1().getY();
 maxyf = getpl1().getY();

 float p2y = getpl2().getY();
 float p3y = getpl3().getY();

 if (p2y>maxyf) maxyf= p2y;
 if (p2y<minyf) minyf = p2y;
```

```
if (p3y>maxyf) maxyf =p3y;
if (p3y<minyf) minyf = p3y;

if(minyf>0){
 int i3;
 for(i3=(int) (minyf) +1;i3%coarse != 0;i3++){}
 miny = i3;
 for(;i3<(int) (maxyf)+1;i3=i3+coarse){}
 maxy = i3-coarse;
}
if(minyf<0 && maxyf>0){
 int j3;
 for(j3=(int) (minyf);j3%coarse != 0;j3++){}
 miny = j3;
 for(;j3<(int) (maxyf)+1;j3=j3+coarse){}
 maxy = j3-coarse;
}
if(minyf<0 && maxyf<0){

 int k3;
 for(k3=(int) (minyf);k3%coarse != 0;k3++){}
 miny = k3;
 for(;k3<(int) (maxyf);k3=k3+coarse){}
 maxy = k3-coarse;
}


int rangey = (maxy - miny)/coarse +1;

float listofvals[2][10]= {0};



for(int i=0;i<3;i++){
 list<point> lpoints;
 larray[i].getGridCrossings(lpoints,coarse);
 list<point>::iterator iter;

 for(iter = lpoints.begin();iter != lpoints.end();iter++){
  point p(*iter);


  float * f1 = & listofvals[0][(int) (((int) p.getY() - miny)/coarse)];
  float * f2 = & listofvals[1][(int) (((int) p.getY() - miny)/coarse)];

  if(*f1 == 0) *f1 = p.getX();
  else{
   if(*f1 <p.getX()) *f2 = p.getX();
   else {
    *f2 = *f1;
    *f1 = p.getX();
   }
  }
 }
```

```
}
//find the x modulo coarse

for (int listofvalscount =0;listofvalscount<rangey;listofvalscount++){



  if (listofvals[0][listofvalscount]>0){

    int i4;
    for(i4 = (int)listofvals[0][listofvalscount]+1;i4%coarse !=0;i4++){}

    for (int xval = i4;
        xval < (int) listofvals[1][listofvalscount] +1; xval=xval+coarse){
      returnpoints.push_back(point (xval,(listofvalscount*coarse) + miny,0));
    }
  }
  else if (listofvals[0][listofvalscount]<0 &&
         listofvals[1][listofvalscount]>0){
    int i5;
    for(i5 = (int) listofvals[0][listofvalscount];i5%coarse!=0;i5++){}

    for(int xval = i5;
       xval<(int) listofvals[1][listofvalscount] +1; xval=xval+coarse){
      returnpoints.push_back(point (xval,(listofvalscount*coarse) + miny,0));
    }
  }
  else{
    int i6;
    for(i6 = (int) listofvals[0][listofvalscount];i6%coarse!=0;i6++){}

    for(int xval = i6;
       xval <(int) listofvals[1][listofvalscount]; xval=xval+coarse){
      returnpoints.push_back(point (xval,(listofvalscount*coarse) + miny,0));
    }
  }
 }
}



//Another Application specific plane segment function...
//This function provides a list of Simplex Lines in a given direction
//that intersect this.


void plane_segment::whichSimplexLinesIntersect(list<line_segment> &llist,int direction,int coarse){



 plane_segment p(project(direction));
 list<point> lpoints;
```

```
 p.whichPointsAreIn(lpoints,coarse);
 list<point>::iterator iter;
 for(iter = lpoints.begin();iter!=lpoints.end();iter++)
  llist.push_back(line_segment (direction,*iter));
}



/*int plane_segment::operator==(const plane_segment &p) const{
 if ((getpl1() == p.getpl1()) && (getpl2() == p.getpl2()) && (getpl3 ==p.getpl3()))
  return 1;
 return 0;
}
*/




class comparepoints{
public:
 bool operator()(point,point);
};

bool comparepoints::operator()(point a,point b){
 if (a<b) return true;
 return false;
}









/*

main(){




 //plane_segment planeman(point(5.9,3.31,2.345),point(0.7,6.82,2.3),point(2.34,1.74,2.1));

 // plane_segment planeman(point(10.543,11.26,3.55),point(15.21,13.47,5.93),
 //         point(19.42,10.34,2.18));
```

```
// plane_segment planeman(point(10.543,11.26,3.55),point(15.21,13.47,5.93),
//    point(12.23,12.76,8.03));

// plane_segment planeman(point(15.21,13.47,5.93), point(19.42,10.34,2.18),
//         point(12.23,12.76,8.03));

// plane_segment planeman(point(10.543,11.26,3.55),point(19.42,10.34,2.18),
//         point(12.23,12.76,8.03));


// plane_segment planeman(point(-0.746,0.435,0),point(0.7678,0.412,0),point(0.102,3.89,0));

 plane_segment planeman(point(10.543,11.26,3.55),point(13.21,13.47,5.93),
  point(15.42,10.34,2.18));

// plane_segment planeman(point(10.543,11.26,3.55),point(13.21,13.47,5.93),
//    point(12.23,12.76,8.03));

// plane_segment planeman(point(13.21,13.47,5.93), point(15.42,10.34,2.18),
//         point(12.23,12.76,8.03));

//plane_segment planeman(point(10.543,11.26,3.55),point(15.42,10.34,2.18),
//    point(12.23,12.76,8.03));


 int direction;
 //for( direction=0;direction<7;direction++){
  direction =1;
   cout<<"\n\n"<<direction<<"\n\n";
   list<line_segment> l4list;
   planeman.whichSimplexLinesIntersect(l4list,direction);


   list<line_segment>::iterator iter1;

   for(iter1=l4list.begin();iter1!=l4list.end();iter1++){
     cout<<"\n\n"<<planeman.intersectsLine(*iter1);

  }
}


*/
```

Appendix B

Image Manipulation Library and Source Code

B.1 Header File (Image.h)

```
//Image.cpp  This file keeps the image data, and allows access to it...


#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "3dgeometry.h"


#ifndef b_a

#define b_a



template<class C> class Image{

 C  matrix[30][30][30];

public:

 Image(char *);
 Image(const Image<C> &);
 Image();
 void displaymatrix15();
 Image<float> getVolumeGradient();
 void setMatrixElement(int,int,int,C);
 C getMatrixElement(int,int,int);
 float getPointSpeed(const point&)const;

};


#endif
```

B.2 Source Code (Image.cpp)

```
//Image.cpp  This file keeps the image data, and allows access to it...
```

```
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include "3dgeometry.h"
#include "Image.h"



template<class C> Image<C>::Image(){

 for(int i =0;i<30;i++){
  for(int j =0;j<30;j++){
   for(int k =0;k<30;k++){

    matrix[i][j][k] = 0;
   }
  }
 }
}

template<class C>Image<C>::Image(const Image<C> & I){


 for(int i=0;i<30;i++){
  for(int j=0;j<30;j++){

   for(int k=0;k<30;k++){

    matrix[i][j][k] = I.matrix[i][j][k];
   }
  }
 }
}



template<class C>Image<C>::Image(char * filename){

 ifstream from(filename);

 C ch;


 for(int i=0;from.get(ch);i++){

  int a,b,c;
  a = i/900;
  b = (i%900)/30;
  c = (i%900)%30;
  matrix[c][b][a] = ch;
 }
```

```
}

template<class C> float Image<C>::getPointSpeed(const point &p) const{

 int x1,y1,z1,x2,y2,z2;

 x1 = (int) p.getX();
 y1 = (int) p.getY();
 z1 = (int) p.getZ();

 x2 = x1+1;
 y2 = y1+1;
 z2 = z1+1;

 point p1(x1,y1,z1);
 point p2(x1,y1,z2);
 point p3(x1,y2,z1);
 point p4(x1,y2,z2);
 point p5(x2,y1,z1);
 point p6(x2,y1,z2);
 point p7(x2,y2,z1);
 point p8(x2,y2,z2);

 float dist1,dist2,dist3,dist4,dist5,dist6,dist7,dist8;

 dist1 = distanceBetweenPoints(p1,p);
 dist2 = distanceBetweenPoints(p2,p);
 dist3 = distanceBetweenPoints(p3,p);
 dist4 = distanceBetweenPoints(p4,p);

 dist5 = distanceBetweenPoints(p5,p);
 dist6 = distanceBetweenPoints(p6,p);
 dist7 = distanceBetweenPoints(p7,p);
 dist8 = distanceBetweenPoints(p8,p);

 return (getMatrixElement(x1,y1,z1)*(3-dist1) +
       getMatrixElement(x1,y1,z2)*(3-dist2) +
       getMatrixElement(x1,y2,z1)*(3-dist3) +
       getMatrixElement(x1,y2,z2)*(3-dist4) +
       getMatrixElement(x2,y1,z1)*(3-dist5) +
       getMatrixElement(x2,y1,z2)*(3-dist6) +
       getMatrixElement(x2,y2,z1)*(3-dist7) +
       getMatrixElement(x2,y2,z2)*(3-dist8));

}



template<class C> void Image<C>::setMatrixElement(int i,int j,int k,C val){
 matrix[i][j][k] = val;
}
```

```
template<class C> C Image<C>::getMatrixElement(int i,int j,int k){

 return matrix[i][j][k];
}



template<class C> void Image<C>::displaymatrix15(){


 cout<<"\n\nImage\n\n";
 for(int k=0;k<30;k++){
  cout<<"\n\nSlice: "<<k<<"\n\n";
  for(int j=0;j<30;j++){
   for(int i=0;i<30;i++){

    cout << (int) getMatrixElement(i,j,k)<<"  ";
   }
   cout <<"\n";
  }

 }

}


template<class C> Image<float> Image<C>::getVolumeGradient(){

 Image<float> x;
 int i,j,k;

 for(i= 1;i<29;i++){

  for(j = 1;j<29;j++){

   for(k =1;k<29;k++){


    C p;
    p = matrix[i][j][k];

    float sum1,sum2,sum3;

    sum1 = pow((p - matrix[i-1][j-1][k+1]),2) +pow((p - matrix[i][j-1][k+1]),2) + pow((p - matrix[i+1][j-
1][k+1]),2) +
     pow((p - matrix[i-1][j][k+1]),2) + pow((p-matrix[i][j][k+1]),2) + pow((p-matrix[i+1][j][k+1]),2)+
     pow((p - matrix[i-1][j+1][k+1]),2)+ pow((p-matrix[i][j+1][k+1]),2)+ pow((p-matrix[i+1][j+1][k+1]),2);


    sum2 = pow((p - matrix[i-1][j-1][k]),2) + pow((p - matrix[i][j-1][k]),2) + pow((p - matrix[i+1][j-1][k]),2) +
     pow((p - matrix[i-1][j][k]),2) + pow((p-matrix[i+1][j][k]),2) + pow((p - matrix[i-1][j+1][k]),2)+
     pow((p-matrix[i][j+1][k]),2) + pow((p-matrix[i+1][j+1][k]),2);
```

```
        sum3 = pow((p - matrix[i-1][j-1][k-1]),2) + pow((p - matrix[i][j-1][k-1]),2) + pow((p - matrix[i+1][j-1][k-
1]),2) +
          pow((p - matrix[i-1][j][k-1]),2) + pow((p-matrix[i][j][k-1]),2) + pow((p-matrix[i+1][j][k-1]),2) +
          pow((p - matrix[i-1][j+1][k-1]),2) + pow((p-matrix[i][j+1][k-1]),2) + pow((p-matrix[i+1][j+1][k-1]),2);


        x.setMatrixElement(i,j,k,sqrt(sum1+sum2+sum3));

      }
     }
    }

    return x;
   }







   /*

   main(){



    Image<char> I("Volume.1");

    I.displaymatrix15();

    Image<float>  x(I.getVolumeGradient());

    cout<<"\n";

    x.displaymatrix15();




   }

   */
```

Appendix C

3D Surface Library and Source Code

C.1 Header File (3dmodel.h)

```
/*** 3-D Surface Model Library... To be used with Segmentation Algorithm****/
/*** Author: Varouj A. Chitilian****/
/*** Date: Nov 10, 1998***/
/*** Email: vchitil@mit.edu,vchitilian@sarnoff.com**/
/*** The model is assumed to be a non directed graph.**/
/*** and so I will use an adjacency list to describ it***/
/*** The Adjacency list will have the following form***/
//
/***  Point A -->  {list of all points A is adjacent to and itself}***/
/***  Point B -->  {list of all points B is adjacent to and itself} ***/
/***   .                    .                ***/
/***   .                    .                ***/
/***   .                    .                ***/
/***   .                    .                ***/


/** Notes:  1) the point itself is included in the adjacency list for ***/
/**          algorithmic convenience... This will be discussed further***/
/**          later on.**/
//
/**        2) The points appearing in the first column will be in order ***/
/**          vertically as described by the ordering system in the file***/
/**          3dgeometry.cpp. The points in a given adjacency list will**/
/**          also be in the same order. The ordering is done so that we **/
/**          can go through each point on the surface in a logical way. **/



#include <stdlib.h>
#include <iostream.h>
#include <iterator.h>
#include <map.h>
#include <set.h>
#include <algorithm>
#include "3dgeometry.h"
#include "Image.h"


/*================================================================*/

/* for convenience in defining classes we use descriptive*/
/* names: AdjacentPoints and AdjacencyList to describe  */
```

```
/* the different components of the Graph.              */


typedef set<point,comparepoints> AdjacentPoints;
typedef map<point,AdjacentPoints,comparepoints> AdjacencyList;


/*=========================================================================*/
/*The Find function for list<point>...                  */
/*this function returns an iterator that points to the point p if it is*/
/*in the AdjacentPoints or returns an iterator that points to the end of*/
/*the adjacent points if the point is not in the AdjacentPoints.      */


AdjacentPoints::iterator setfind(point p,AdjacentPoints & k);


/*=========================================================================*/
/* The Surface Class is defined as a derived class from Adjacency List*/
/* This allows us to expand the map class to include operations such as*/
/* applying a displacement to a surface, reparameterizing a surface, and*/
/* performing Topological Transformations on a surface.              */


class Surface: public AdjacencyList{
  map<point,bool,comparepoints> IsIn;
  float TotalSpeed;
  float ContourSpeed;
  int NumberOfPlanes;
  void replaceAwithB(point,point);
  point getOutwardNormal(const plane_segment &,float);
  void updateTotalSpeed(const point&);
  void setContourSpeed();

public:

  map<point,bool,comparepoints> getIsIn();
  int IsInSize();
  int getNumPlanes();
  int getConnComps();

  void incNumPlanes();
  class DuplicatePointException{};
  class SurfaceOutOfBoundsException{};
  point displacePoint(point,float) throw (SurfaceOutOfBoundsException);

  Surface();
  void applySpeed(const Image<float> &) throw (DuplicatePointException,SurfaceOutOfBoundsException);
  void insert(const plane_segment &);

  point approximateNormal(const point&,float);
```

```
 float approximateCurvaturemag(const point&);
 int curvatureSign(const point&,const point&);
 float getCurvature(const point&);
 void writeVTKModel(char *);
 void setIsIn(const map<point,bool,comparepoints>);
 point getCenterOfMass();
 float getContourSpeed();

};



/*=========================================================================*/
/* Class PatchIterator is used to access all planes that are adjacent to*/
/* a certain point                                                         */
/*=========================================================================*/

class PatchIterator{

 Surface surface;
 Surface::iterator p1;
 AdjacentPoints::iterator p2,p3;
 bool Last;

public:

 class NoMoreException{};
 PatchIterator(const Surface &, const point &);
 plane_segment getNextPlane() throw (NoMoreException);

};



/*=========================================================================*/
/* CLASS PLANEITERATOR is used to access planes that make up a surface  */
/* in an orderly Fashion.                                                  */

 class PlaneIterator{

 Surface surface;

 Surface::iterator p1;

 AdjacentPoints::iterator p2,p3;

 bool Last;

public:
 class NoMoreException{};
 class SurfaceNotInitException{};
 PlaneIterator(const Surface &) throw (SurfaceNotInitException);
 plane_segment getNextPlane() throw (NoMoreException);
```

};

C.2 Source Code (3dmodel.cpp)

```
/*** 3-D Surface Model Library... To be used with Segmentation Algorithm****/
/*** Author: Varouj A. Chitilian****/
/*** Date: Nov 10, 1998***/
/*** Email: vchitil@mit.edu,vchitilian@sarnoff.com**/
/*** The model is assumed to be a non directed graph.**/
/*** and so I will use an adjacency list to describ it***/
/*** The Adjacency list will have the following form***/
//
/***  Point A -->  {list of all points A is adjacent to and itself}***/
/***  Point B -->  {list of all points B is adjacent to and itself} ***/
/***  .                    .                    ***/
/***  .                    .                    ***/
/***  .                    .                    ***/
/***  .                    .                    ***/


/** Notes:  1) the point itself is included in the adjacency list for ***/
/**          algorithmic convenience... This will be discussed further***/
/**          later on.**/
//
/**        2) The points appearing in the first column will be in order ***/
/**          vertically as described by the ordering system in the file***/
/**          3dgeometry.cpp. The points in a given adjacency list will**/
/**          also be in the same order. The ordering is done so that we **/
/**          can go through each point on the surface in a logical way. **/



#include <stdlib.h>
#include <math.h>
#include <iostream.h>
#include <fstream.h>
#include <iterator.h>
#include <map.h>
#include <set.h>
#include <algorithm>
#include "3dgeometry.h"
#include "Image.cpp"



/*==========================================================================*/
```

```
/* for convenience in defining classes we use descriptive*/
/* names: AdjacentPoints and AdjacencyList to describe  */
/* the different components of the Graph.              */


typedef set<point,comparepoints> AdjacentPoints;
typedef map<point,AdjacentPoints,comparepoints> AdjacencyList;


/*=====================================================================*/
/*The Find function for list<point>...                                 */
/*this function returns an iterator that points to the point p if it is*/
/*in the AdjacentPoints or returns an iterator that points to the end of*/
/*the adjacent points if the point is not in the AdjacentPoints.       */



AdjacentPoints::iterator setfind(point p,AdjacentPoints & k){
 AdjacentPoints::iterator i;
 for(i=k.begin();i!=k.end(); i++){
  if ((*i) == p) return i;
 }
 return i;
}




/*=====================================================================*/
/* The Surface Class is defined as a derived class from Adjacency List*/
/* This allows us to expand the map class to include operations such as*/
/* applying a displacement to a surface, reparameterizing a surface, and*/
/* performing Topological Transformations on a surface.                */



class Surface: public AdjacencyList{

 map<point,bool,comparepoints> IsIn;
 float TotalSpeed;
 float ContourSpeed;
 int NumberOfPlanes;
 void replaceAwithB(point,point);
 point getOutwardNormal(const plane_segment &,float);
 void updateTotalSpeed(const point&);
 void setContourSpeed();
public:

 map<point,bool,comparepoints> getIsIn();

 int IsInSize();
 int getNumPlanes();
 int getConnComps();
```

```
 //Watch out this public function should not be abused...
 void incNumPlanes();


 class DuplicatePointException{};
 class SurfaceOutOfBoundsException{};
 point displacePoint(point,float) throw (SurfaceOutOfBoundsException);

 Surface();
 void applySpeed(const Image<float>&)
  throw (DuplicatePointException,SurfaceOutOfBoundsException);
 void insert(const plane_segment &);

 point approximateNormal(const point &,float);
 float approximateCurvaturemag(const point&);
 int curvatureSign(const point&,const point&/*this is the nomal vector*/);
 float getCurvature(const point&);
 void writeVTKModel(char *);
 void setIsIn(const map<point,bool,comparepoints>);
 point getCenterOfMass();
 float getContourSpeed();
};



map<point,bool,comparepoints> Surface::getIsIn(){
 return IsIn;
}

void Surface::updateTotalSpeed(const point & p){
 TotalSpeed = TotalSpeed + magnitude(p);
}

void Surface::setContourSpeed(){

 ContourSpeed = TotalSpeed/size();
}

float Surface::getContourSpeed(){
 return ContourSpeed;
}


point Surface::getCenterOfMass(){
 Surface::iterator iter;

 point p(0,0,0);

 for (iter = (*this).begin();iter != (*this).end();iter++){
  p = p + (*iter).first;
 }

 int sz = (*this).size();
```

```
 return (p/sz);
}

int Surface::IsInSize(){
 return IsIn.size();
}


void Surface::setIsIn(const map<point,bool,comparepoints> IsInTop){
 IsIn = IsInTop;
}



Surface::Surface():AdjacencyList(){

 NumberOfPlanes =0;
 TotalSpeed =0;
}


int Surface::getConnComps(){
 int x;
 x=0;

 Surface::iterator iter;
 set<point,comparepoints>::iterator iter2;

 for(iter = (*this).begin();iter!=(*this).end();iter++){
  iter2 = ((*iter).second).end();
  iter2--;
  if((*iter).first == (*iter2))
    x++;
 }
 return x;
}


int Surface::getNumPlanes(){
 return NumberOfPlanes;
}

void Surface::incNumPlanes(){
 NumberOfPlanes++;
}




/*************************************************************************/
/*The function displacePoint calculates the evolution of a single point*/
```

```
/*on the surface.  Using information about image gradients as well as  */
/* information about local surface features.                          */
/**********************************************************************/



point Surface::displacePoint(point P,float speed) throw (SurfaceOutOfBoundsException){
 float c = approximateCurvaturemag(P);
 int ispeed = (int) speed;
 float s = 800;
 if (ispeed == 0) s =1;
 if (ispeed == 1) s =0.5;
 if (ispeed == 2) s =0.2;
 if (ispeed == 3) s =0.1;
 if (ispeed >  3) s = 0.05;

 if (s ==800){ cout<<"problem 800... exiting";
 cout<<"\n speed value: "<<speed<<"\nispeed: "<<ispeed<<"\n";
 exit(0);
 }
 point Q(normalize(approximateNormal(P,c))*getCurvature(P)*s + P);

 if (Q.getX()>28 ||Q.getY()>28||Q.getZ()>28) throw SurfaceOutOfBoundsException();
 return Q;
}

/*replaceAwithB will replace the point A in the surface model with the*/
/*point B, but will keep all connectivity information the same.      */
/*in order to change a single value first we must first copy the list */
/*with a different key value... we then change the old values in all of the*/
/*lists in the map.  This is used when evolving the surface.*/

void Surface::replaceAwithB(point A,point B){

 AdjacentPoints::iterator original;
 AdjacentPoints set1;
 AdjacentPoints set2;



 set1 = (*(find(A))).second;

 copy(set1.begin(),set1.end(),inserter(set2,set2.begin()));

 set2.erase(A);
 set2.insert(B);
 erase(A);

 (*this)[B] =set2;


 AdjacentPoints::iterator fixtherest;
 AdjacentPoints set3;
```

```
 set3 = (*(find(B))).second;


 for(fixtherest=set3.begin();fixtherest!=set3.end();fixtherest++){

  if(*fixtherest != B){
    AdjacentPoints set4;
    set4 = (*(find(*fixtherest))).second;
    set4.erase(A);
    set4.insert(B);
    (*this)[*fixtherest] = set4;
   }
 }

}

/*************************************************************************/
/*Insert: This function inserts the points of a plane appropriately into*/
/*      the surface.  Therefore it takes the first point, sees if it  */
/*      is in the surface yet or not.  If it is, updates that points  */
/*      adjacency list to include the other two points of the plane   */
/*      if not adds the point and puts the other two points as members*/
/*      the adjacency list.  It then does the same thing for the      */
/*      two points.                                                   */
/*************************************************************************/

void Surface::insert(const plane_segment & plane){

 //cout<<"\n INSERTING PLANE"<<plane;

 point p1(plane.getpl1());
 //cout<< "\nINSERTING POINT "<<p1;
 point p2(plane.getpl2());
 //cout<< "\nINSERTING POINT "<<p2;
 point p3(plane.getpl3());
 //cout<< "\nINSERTING POINT "<<p3;

 ((*this)[p1]).insert(p1);
 ((*this)[p1]).insert(p2);
 ((*this)[p1]).insert(p3);


 ((*this)[p2]).insert(p1);
 ((*this)[p2]).insert(p2);
 ((*this)[p2]).insert(p3);

 ((*this)[p3]).insert(p1);
 ((*this)[p3]).insert(p3);
 ((*this)[p3]).insert(p2);


}
```

```
/*************************************************************************/
/*Apply Speed: This function applies the appropriate speed to each point on */
/*the surface.                                                           */
/*Things to watch out for:  Two points moving to the same location, A moving*/
/*to B while B moves to C.                                               */
/*************************************************************************/

void Surface::applySpeed(const Image<float> &I) throw (DuplicatePointException,SurfaceOutOfBoundsException){

 int numpoints;
 numpoints = size();


 point * pointarray = new point[numpoints];
 point * newpointarray = new point[numpoints];

 AdjacencyList::iterator yo;
 int i = 0;

 for(yo = (*this).begin(); yo!= (*this).end(); yo++, i++){
  point q((*yo).first);
  pointarray[i] = q;
  float speed = I.getPointSpeed(q);
  point z;
  try{
    z = displacePoint(q,speed);
  }
  catch (SurfaceOutOfBoundsException){
    throw SurfaceOutOfBoundsException();
  }
  newpointarray[i] = z;
  updateTotalSpeed(z);

 }

 for(i=0;i<numpoints;i++){
  replaceAwithB(pointarray[i],newpointarray[i]);
 }
 setContourSpeed();
 delete pointarray;
 delete newpointarray;
}
```

```
/*=======================================================================*/
/* Class PatchIterator is used to access all planes that are adjacent to*/
```

```
/* a certain point                                        */
/*=======================================================================*/

class PatchIterator{

 Surface surface;
 Surface::iterator p1;
 AdjacentPoints::iterator p2,p3;
 bool Last;

public:

 class NoMoreException{};
 PatchIterator(const Surface &, const point &);
 plane_segment getNextPlane() throw (NoMoreException);

};

PatchIterator::PatchIterator(const Surface & surf,const point & p){

 surface = surf;
 p1 = surface.find(p);
 p2 = ((*p1).second).begin();
 p3 =p2;

}


plane_segment PatchIterator::getNextPlane() throw (NoMoreException){

 while (p2 != ((*p1).second).end()){
  AdjacentPoints::iterator p4;

  p4 = p2;
  p4++;
  if((*p2) == (*p1).first && p4 == ((*p1).second).end()) break;

  if((*p2) == (*p1).first){
  p2++;
  p3++;
  }

   p3++;

  while(p3 != ((*p1).second).end()){

   AdjacentPoints::iterator i;
   i = setfind(*p3,(surface[*p2]));
   if(i== (surface[*p2]).end() || (*p3)==(*p1).first)
    p3++;
   else return plane_segment((*p1).first,*p2,*p3);
  }
  p2++;
  p3 =p2;
```

```
  }

  throw NoMoreException();

}




/*=========================================================================*/
/* CLASS PLANEITERATOR is used to access planes that make up a surface  */
/* in an orderly Fashion.                                                */

class PlaneIterator{

  Surface surface;

  Surface::iterator p1;

  AdjacentPoints::iterator p2,p3;

  bool Last;

public:
  class NoMoreException{};
  class SurfaceNotInitException{};
  PlaneIterator(const Surface &) throw (SurfaceNotInitException);
  plane_segment getNextPlane() throw (NoMoreException);


};




//The constructor will throw a SurfaceNotInitException, when either:
//the surface has not yet been filled or the surface has incorrect
//format and the first element of adjacency list is not the same as
//the first key in the map

PlaneIterator::PlaneIterator(const Surface & s) throw (SurfaceNotInitException){
  surface =s;
  p1 = surface.begin();
  if (p1 != surface.end()){
    p2 = ((*p1).second).begin();
    if (p2 != ((*p1).second).end()){
      p2++;
      p3 = p2;
    }
    else
      throw SurfaceNotInitException();
  }
  else
```

```
  throw SurfaceNotInitException();
}



/*The function getNextPlane() iterates through the planes on a surface*/
/*in an efficient manner. The iteration consists of moving through the*/
/*adjacency list in an orderly fashion, and identifying the planes    */
/*without duplication. Everytime it is called it will return the next*/
/*plane. This iterator will be used during reparameterization of the */
/*surface.                                                           */

plane_segment PlaneIterator::getNextPlane() throw (NoMoreException){
 //list iterators p2 and p3 should be at same location when function is
 //entered.
 while(1){
  while (p2 != ((*p1).second).end()){
   p3++;
   while(p3 != ((*p1).second).end()){
    AdjacentPoints::iterator i;
    i = setfind(*p3,(surface[*p2]));
    if(i== (surface[*p2]).end())
     p3++;
    else return plane_segment((*p1).first,*p2,*p3);
   }
   p2++;
   p3 =p2;
  }
  p1 ++;
  if (p1 != surface.end()){
   p2 = setfind((*p1).first,surface[(*p1).first]);
   p2++;
   p3=p2;
  }
  else throw NoMoreException();
 }
}



point Surface::getOutwardNormal(const plane_segment & ps,float curv){

 //bool n11,n12,n13,n21,n22,n23;
 int n1,n2,n3;
 point p1(ps.getpl1());
 point p2(ps.getpl2());
 point p3(ps.getpl3());

 point CM = getCenterOfMass();
```

```
if (curv>0.5){

 n1 = (distanceBetweenPoints((normalize(ps.getNormal1())*0.5+p1),CM)>
     distanceBetweenPoints((normalize(ps.getNormal2())*0.5+p1),CM));

 n2 = (distanceBetweenPoints((normalize(ps.getNormal1())*0.5+p2),CM)>
     distanceBetweenPoints((normalize(ps.getNormal2())*0.5+p2),CM));

 n3 = (distanceBetweenPoints((normalize(ps.getNormal1())*0.5+p3),CM)>
     distanceBetweenPoints((normalize(ps.getNormal2())*0.5+p3),CM));


 /* n11 = IsIn[(normalize(ps.getNormal1())*0.5+p1).closestPoint()];
 n12 = IsIn[(normalize(ps.getNormal1()*0.5)+p2).closestPoint()];
 n13 = IsIn[(normalize(ps.getNormal1())*0.5+p3).closestPoint()];
 n21 = IsIn[(normalize(ps.getNormal2())*0.5+p1).closestPoint()];
 n22 = IsIn[(normalize(ps.getNormal2())*0.5+p2).closestPoint()];
 n23 = IsIn[(normalize(ps.getNormal2())*0.5+p3).closestPoint()];
 */


}
else{
 n1 = (distanceBetweenPoints((normalize(ps.getNormal1())+p1),CM)>
     distanceBetweenPoints((normalize(ps.getNormal2())+p1),CM));

 n2 = (distanceBetweenPoints((normalize(ps.getNormal1())+p2),CM)>
     distanceBetweenPoints((normalize(ps.getNormal2())+p2),CM));

 n3 = (distanceBetweenPoints((normalize(ps.getNormal1())+p3),CM)>
     distanceBetweenPoints((normalize(ps.getNormal2())+p3),CM));




 /*n11 = IsIn[(normalize(ps.getNormal1())+p1).closestPoint()];
 n12 = IsIn[(normalize(ps.getNormal1())+p2).closestPoint()];
 n13 = IsIn[(normalize(ps.getNormal1())+p3).closestPoint()];
 n21 = IsIn[(normalize(ps.getNormal2())+p1).closestPoint()];
 n22 = IsIn[(normalize(ps.getNormal2())+p2).closestPoint()];
 n23 = IsIn[(normalize(ps.getNormal2())+p3).closestPoint()];
 */
 }

// int n1 = n11+n12+n13;

//int n2 = n22+n23+n21;
/*
if (n1==0& n2 ==0){
 cout << "Problem: no normals within surface... exiting";
 exit(0);
}
```

```
*/

int sum = n1+n2+n3;

if (sum >=2){
 //cout<<"\nYES\n";

 return ps.getNormal1();
}
else{
 //cout<<"\nNO\n";

 return ps.getNormal2();
}
/*
if(n1==n2) cout<<"\n\nYO\n\n";

if (n1>=n2)
 return ps.getNormal1();
else{
 return ps.getNormal2();
 cout << "\nME\n";

}
*/
}
```

```
//fortunately we only need the curvature magnitude to get a good
//approximation to the normal...
//So we can use this normal to find the sign of the curvature.

int Surface::curvatureSign(const point& p, const point & normal){

 PatchIterator PI((*this),p);

 plane_segment pl(PI.getNextPlane());

 point p1(pl.getpl1());
 point p2(pl.getpl2());
 point p3(pl.getpl3());

 point ptest(0,0,0);
 if (p1 == p)
  ptest = p2;
 if (p2 == p)
  ptest = p1;
 if (p3 == p)
```

```
  ptest = p1;

 point vector(ptest-p);

 float angle = acos((dot(vector,normal))/(magnitude(vector)*magnitude(normal)));

 if (angle >3.14/2){
  return -1;
 }
 else return 1;


}

float Surface::approximateCurvaturemag(const point &p){

 PatchIterator PI((*this),p);

 float TotalAngle =0;


 while(1){
  try{

    plane_segment pl(PI.getNextPlane());



    point a(pl.getpl1());
    point b(pl.getpl2());
    point c(pl.getpl3());

    point vectorp1;

    point vector1p2;

    point vector2p2;


    if (a == p){
      vectorp1 = a;
      vector1p2 = b;
      vector2p2 =c;
    }
    if (b == p){
      vectorp1 =b;
      vector1p2 = a;
      vector2p2 = c;
    }
    if (c == p){
      vectorp1 =c;
      vector1p2 = a;
      vector2p2 = b;
    }
```

```
    point v1(vector1p2 - vectorp1);
    point v2(vector2p2 - vectorp1);

    float f117;

    f117 = angleBetweenVectors(v1,v2);

    //cout<<"\n Plane is: "<<pl;
    //cout <<"\n Angle: " <<f117;

    TotalAngle = TotalAngle + f117;
   }
   catch (PatchIterator::NoMoreException){break;}
  }

 return (1 - TotalAngle);
}



point Surface::approximateNormal(const point& p, float curve){

 PatchIterator PI((*this),p);

 float sumx,sumy,sumz;

 sumx = 0;
 sumy = 0;
 sumz = 0;

 int num =0;

 while(1){
  try{
    plane_segment pl(PI.getNextPlane());
    point temp(getOutwardNormal(pl,curve));
    num++;

    sumx = sumx + temp.getX();
    sumy = sumy + temp.getY();
    sumz = sumz + temp.getZ();
   }
   catch (PatchIterator::NoMoreException){break;}
  }

 if (num ==0){
  cout<<"\n\nDIVIDING BY ZERO THIS SHOULD NOT HAPPEN";
  exit(0);
 }


 //We are approximating the normal at a point by the component based
 //average of its adjacent planes.
```

```
 return point(sumx/(float) num,sumy/(float) num,sumz/(float) num);


}


float Surface::getCurvature(const point& p){
 //this function will return a number between 0 and 2
 //a value of 1 means the surface is flat...
 //a value of 0 means the point is highly extrusive
 //a value of 2 means the point is highly intrusive

 float mag = approximateCurvaturemag(p);

 point normal(approximateNormal(p,mag));

 int sign = curvatureSign(p,normal);

 //return (sign*mag+1);
 if (sign == -1) return (1-mag);
 else return (2*mag +1);
}


/*=================================================================*/
/* MAIN PROGRAM USED TO TEST FEATURES AS THEY ARE DEVELOPED      */
/*=================================================================*/



/*
main(){

 Surface mp;

 point a(0,0,0);
 point b(2,0,0);
 point c(2,2,0);
 point d(0,2,0);
 point e(1,1,100);
 point f(1,1,0);

 plane_segment plane1(a,b,e);
 plane_segment plane2(a,b,f);
 plane_segment plane3(a,d,e);
 plane_segment plane4(a,d,f);
 plane_segment plane5(c,d,e);
 plane_segment plane6(c,d,f);
 plane_segment plane7(b,c,e);
 plane_segment plane8(b,c,f);
```

```
  plane_segment plane1(point(10.543,11.26,3.55),point(13.21,13.47,5.93),
        point(15.42,10.34,2.18));

 plane_segment plane2(point(10.543,11.26,3.55),point(13.21,13.47,5.93),
        point(12.23,12.76,8.03));

 plane_segment plane3(point(13.21,13.47,5.93), point(15.42,10.34,2.18),
        point(12.23,12.76,8.03));

 plane_segment plane4(point(10.543,11.26,3.55),point(15.42,10.34,2.18),
        point(12.23,12.76,8.03));


 mp.insert(plane1);
 mp.insert(plane2);
 mp.insert(plane3);
 mp.insert(plane4);
 mp.insert(plane5);
 mp.insert(plane6);
 mp.insert(plane7);
 mp.insert(plane8);




 float angle1;
 float angle2;
 angle1 = mp.approximateCurvaturemag(e);
 angle2 = mp.approximateCurvaturemag(f);



 cout<<"\nThe Angle at point e is: "<< angle1 <<"\n";
 cout<<"\nThe Angle at point f is: "<< angle2 <<"\n";

 cout<<"\nThe sign at point e is: "<<

 }
 */




/*************************************************************************/
/*writeVTKModel will write the model to a file that is readable by the VTK*/
/*visualization Tool Kit.                                                */
/*************************************************************************/

void Surface::writeVTKModel(char * filename){
```

```
ofstream VTKoutput(filename);
if(!VTKoutput){
 cout << "Cannot open file "<< filename;
 exit(0);
}
   VTKoutput<<"# vtk DataFile Version 2.0\nvtk output\nASCII\nDATASET POLYDATA\nPOINTS
"<<(*this).size()<<" float\n";
AdjacencyList::iterator yo;

for(yo= (*this).begin();yo!= (*this).end();yo++){

 VTKoutput<<((*yo).first).getX()<<" "<<((*yo).first.getY())<<" "
      <<((*yo).first).getZ()<<"\n";
}
VTKoutput<<"POLYGONS "<<(*this).getNumPlanes()<<" "
     <<((*this).getNumPlanes())*4<<"\n";

PlaneIterator PI(*this);

point parray[3];
int intarray[3];

AdjacencyList::iterator iter;

while(1){
 try{
   plane_segment pl(PI.getNextPlane());
   parray[0] = pl.getpl1();
   parray[1] = pl.getpl2();
   parray[2] = pl.getpl3();
   int k=0;
   for(int i=0;i<3;i++){
    int j =0;

    for(iter = (*this).begin();iter !=(*this).end();iter++,j++){
     if((*iter).first == parray[i]){
       intarray[k] = j;
       k++;
       break;
      }
     }
   }
   VTKoutput<<"3 "<<intarray[0]<<" "<<intarray[1]<<" "<<intarray[2]<<"\n";
 }
 catch(PlaneIterator::NoMoreException){break;}



}

}
```

Appendix D

3D Simplex Library and Source Code

D.1 Header File (3dsimplex.h)

```
/*** 3-D Simplex Library... To be used with Segmentation Algorithm****/
/*** Author: Varouj A. Chitilian****/
/*** Date: Nov 10, 1998***/
/*** Email: vchitil@mit.edu,vchitilian@sarnoff.com**/
/*** In the 3D segmentation Algorithm, a simplicial grid is used to **/
/*** reparameterize the model of the surface.***/
/*** This file contains two major classes: Simplex Unit and Simplicial Grid***/
//
/***  A simplex unit is simply represented as a set of 19 line segments***/
/***  Unit = {l1,l2,l3,l4....,l19} and implicitly contains 6 simplex cells ***/
/***  A simplicial grid is represented by a width and a height. ***/
/***   It provides the ability to construct a simplex unit anywhere in the***/
/***  virtual space defined by the width and the height...***/
/*** ***/



#include <stdlib.h>
#include <iostream.h>
#include "3dgeometry.h"

class Simplex{

  point location;
  int type;
public:
  Simplex(const point &, int);
  Simplex();
  Simplex(const Simplex & s);
  void operator=(const Simplex&);
  bool operator==(const Simplex&) const;
  point getLocation() const;
  int getType() const;
};

/*=============================================================================*/
/*The class SimplexEdge defines a line segment that can only have one of */
/*7 pre-specified directions.                                                  */
/* Index                Direction                                            */
/* -----                ---------                                            */
/*  0                    i                                                   */
/*  1                    j                                                   */
```

```
/*  2                   k                   */
/*  3                   i+j                 */
/*  4                   j+k                 */
/*  5                   -i+k                */
/*  6                   i+j+k               */


class SimplexEdge:public line_segment{

 int direction;
public:
 SimplexEdge(const line_segment &, int/*coarse*/);
 SimplexEdge(const point &,const point &, int/*coarse*/);
 //SimplexEdge(const point &,const point &,int);
 SimplexEdge(const point &,const line_segment &, int /*coarse*/);
 SimplexEdge(const SimplexEdge &);
 SimplexEdge();
 class NotASimplexEdgeException{};
 int getDirection() const;
 line_segment getLineSegment() const;
 float distanceFromP(const point &,
        int/*point number this can either be 1 or 0*/ );
 void operator=(const SimplexEdge &);
};

struct SimplexTypePointPair{
 int c;
 point pointlist[4];
};


struct EdgeSLPair{
 int NumberIncident;
 Simplex simplexlist[6];
};
/*=========================================================================*/
/*Simplicial Grid class holds the information about the size of the Grid*/
/* it also serves up the right simplex unit when it is required and does*/
/* checking to ensure that everything is within the correct bounds.    */

class SimplicialGrid{
 friend class EtoSIterator;
 friend class StoPIterator;
 int xsize;
 int ysize;
 int zsize;
 int coarseness;



 struct SimplexTypePointPair TypetoPoint[6];
 struct EdgeSLPair EdgetoSimplicies[7];

 void initEtoS();
```

```
 void initTtoP();

public:

 SimplicialGrid(int,int,int,int /*coarseness*/);

 class OutofBoundsException{};

 int getXSize() const;
 int getYSize() const;
 int getZSize() const;
 int getCoarse() const;
};



class EtoSIterator{

 int Direction;
 point Location;
 int HowManyLeft;

public:

 EtoSIterator(const SimplexEdge&, const SimplicialGrid&);
 Simplex nextSimplex(const SimplicialGrid&);
 class NoSimpliciesLeftException{};

};


class StoPIterator{

 int TypeOfSimplex;
 point Location;
 int HowManyLeft;

public:

 StoPIterator(const Simplex &);
 point nextPoint(const SimplicialGrid &);
 class NoPointsLeftException{};
};

class SimplexEdgePointPair{

 SimplexEdge s;
 point p;
 void sets(const SimplexEdge &);
 void setp(const point &);

public:

 SimplexEdgePointPair();
```

```
  SimplexEdgePointPair(const point &,const SimplexEdge &);
  SimplexEdgePointPair(const SimplexEdgePointPair &);
  point getPoint() const;
  SimplexEdge getSimplexEdge() const;
};

class comparesimplex{
public:
  bool operator()(Simplex,Simplex);
};



ostream& operator<<(ostream& s,Simplex &Si);

ostream& operator<<(ostream& s,SimplexEdge &se);

ostream& operator<<(ostream& s,SimplexEdgePointPair &Sip);
```

D.2 Source Code (3dsimplex.cpp)

```
/*** 3-D Simplex Library... To be used with Segmentation Algorithm****/
/*** Author: Varouj A. Chitilian****/
/*** Date: Nov 10, 1998***/
/*** Email: vchitil@mit.edu,vchitilian@sarnoff.com**/
/*** In the 3D segmentation Algorithm, a simplicial grid is used to **/
/*** reparameterize the model of the surface.***/
/*** This file contains two major classes: Simplex  and Simplicial Grid***/
//
/***  A simplex is simply represented as a location and a character that***/
/*** describes the whereabouts of the Simplex the bounds of the location***/
/***  The location is simply a point in 3 space and the type is a character***/
/***  such that Type belongs to the set {A,B,C,D,E,F} ***/

/***  A simplicial grid is represented by a width and a height. ***/
/***   It provides the ability to construct a simplex anywhere in the***/
/***  virtual space defined by the width and the height...***/
/*** It also (when initialized) gives information about how the edges of ***/
/*** the simplicies interrelate.  And also gives information about the  ***/
/*** verticies of a given Simplex ***/



#include <stdlib.h>
#include <iostream.h>
#include "3dgeometry.h"


class Simplex{
```

```
 point location;
 int type;
public:
 Simplex(const point &, int);
 Simplex();
 Simplex(const Simplex & s);
 void operator=(const Simplex&);
 bool operator==(const Simplex&) const;
 point getLocation() const;
 int getType() const;
};


Simplex::Simplex(){
 location = point();
 type = 0;
}

int Simplex::getType() const{
 return type;
}

point Simplex::getLocation() const{
 return location;
}

Simplex::Simplex(const Simplex &s){
 location = s.getLocation();
 type = s.getType();
}

Simplex::Simplex(const point & p, int c){

 location = p;
 type = c;
}



void Simplex::operator=(const Simplex & s){
 location = s.getLocation();
 type = s.getType();
}


bool Simplex::operator==(const Simplex & s) const{
 if(getLocation() == s.getLocation() && getType() == s.getType())
   return true;
 return false;
}

/*=========================================================================*/
/*The class SimplexEdge defines a line segment that can only have one of */
/*7 pre-specified directions.                                             */
```

```
/* Index                Direction                */
/* -----                ---------                */
/*  0                    i                       */
/*  1                    j                       */
/*  2                    k                       */
/*  3                    i+j                     */
/*  4                    j+k                     */
/*  5                    -i+k                    */
/*  6                    i+j+k                   */


class SimplexEdge:public line_segment{

 int direction;
public:
 SimplexEdge(const line_segment &, int /*coarseness*/);
 SimplexEdge(const point &,const point &, int /*coarseness*/);
 //SimplexEdge(const point &,const point &,int);
 SimplexEdge(const point &,const line_segment &,int/*coarseness*/);
 SimplexEdge(const SimplexEdge &);
 SimplexEdge();
 class NotASimplexEdgeException{ };
 int getDirection() const;
 line_segment getLineSegment() const;
 float distanceFromP(const point &,
        int/*point number this can either be 1 or 0*/ );
 void operator=(const SimplexEdge &);
};


SimplexEdge::SimplexEdge():line_segment(){
 direction =0;
}



int SimplexEdge::getDirection() const{
 return direction;
}



SimplexEdge::SimplexEdge(const line_segment & l, int coarse):line_segment(l){
 point p(l.getp2() - l.getp1());
 if (p == point(1*coarse,0,0))
  direction =0;
 if (p == point(0,1*coarse,0))
  direction =1;
 if (p == point(0,0,1*coarse))
  direction =2;
 if (p == point(1*coarse,1*coarse,0))
  direction =3;
 if (p == point(0,1*coarse,1*coarse))
  direction =4;
```

```
 if (p == point(-1*coarse,0,1*coarse))
   direction =5;
 if (p == point(1*coarse,1*coarse,1*coarse))
   direction =6;
 else
   throw NotASimplexEdgeException();
}

SimplexEdge::SimplexEdge(const SimplexEdge & se):line_segment(se)
{
 direction = se.getDirection();
}

SimplexEdge::SimplexEdge(const point &p1,const point &p2, int coarse):line_segment(p1,p2){

point p(p2 - p1);

if (p == point(1*coarse,0,0))
  direction =0;
if (p == point(0,1*coarse,0))
   direction =1;
  if (p == point(0,0,1*coarse))
   direction =2;
  if (p == point(1*coarse,1*coarse,0))
   direction =3;
  if (p == point(0,1*coarse,1*coarse))
   direction =4;
  if (p == point(-1*coarse,0,1*coarse))
   direction =5;
  if (p == point(1*coarse,1*coarse,1*coarse))
   direction =6;
  else throw NotASimplexEdgeException();
}
/*
SimplexEdge::SimplexEdge (const point &p1,const point &p2,int dir):
  line_segment(p1,p2){
  if (dir <7 && dir >-1)
   direction = dir;
  else throw NotASimplexEdgeException();
}
*/


SimplexEdge::SimplexEdge(const point &p,const line_segment &l,int coarse){
  point pdiff(l.getp2()-l.getp1());
  int max;
  int dir;
  if(pdiff.getX()>pdiff.getY() && pdiff.getX()>pdiff.getZ())
   max = (int) pdiff.getX();
  else
   if(pdiff.getY()>pdiff.getZ())
   max = (int) pdiff.getY();
   else
     max = (int) pdiff.getZ();
```

```
point ptest(point (pdiff.getX()/max,pdiff.getY()/max, pdiff.getZ()/max));

if(ptest == point(1,0,0)) dir =0;
if(ptest == point(0,1,0)) dir=1;
if(ptest == point(0,0,1)) dir= 2;
if(ptest == point (1,1,0)) dir=3;
if (ptest == point (0,1,1)) dir=4;
if (ptest == point (-1,0,1)) dir=5;
if (ptest == point (1,1,1)) dir=6;

point pstart;
point pend;

switch (dir){

case 0:
 /*
 pstart = point((float) ((int) p.getX()),p.getY(),p.getZ());
 pend = point(pstart + ptest);
 break;
 */


 int i1;
 for (i1 = (int) p.getX(); i1%coarse!=0;i1--){}
 pstart = point((float) i1,p.getY(),p.getZ());
 pend = point(pstart + (ptest*coarse));
 break;

case 1:
 /*
 pstart = point(p.getX(),(float) ((int) p.getY()),p.getZ());
 pend = point(pstart + ptest);
 break;
 */

 int i2;
 for (i2 = (int) p.getY(); i2%coarse!=0;i2--){}
 pstart = point(p.getX(),(float) i2,p.getZ());
 pend = point(pstart+(ptest*coarse));
 break;

case 2:
 /*
 pstart = point(p.getX(),p.getY(),(float) ((int) p.getZ()));
 pend = point(pstart+ptest);
 break;
 */


 int i3;
 for (i3 = (int) p.getZ(); i3%coarse!=0;i3--){}
 pstart = point(p.getX(),p.getY(),(float) i3);
```

```
 pend = point(pstart+(ptest*coarse));
 break;

case 3:
 /*
 pstart = point((float)((int) p.getX()),(float) ((int) p.getY()),p.getZ());
 pend = point(pstart + ptest);
 break;
 */


 int i4;
 int j4;
 for(i4 = (int) p.getX();i4%coarse!=0;i4--){}
 for(j4 = (int) p.getY();j4%coarse!=0;j4--){}
 pstart = point((float) i4,(float) j4,p.getZ());
 pend = point(pstart + (ptest*coarse));
 break;

case 4:
 /*
 pstart = point(p.getX(),(float)((int) p.getY()),(float) ((int) p.getZ()));
 pend = point(pstart + ptest);
 break;
 */


 int i5;
 int j5;
 for(i5 = (int) p.getY();i5%coarse!=0;i5--){}
 for(j5 = (int) p.getZ();j5%coarse!=0;j5--){}

 pstart = point(p.getX(),(float) i5,(float) j5);
 pend = point(pstart + (ptest*coarse));
 break;

case 5:
 /*
 pstart = point((float) ((int) p.getX() +1),p.getY(),(float) ((int) p.getZ()));
 pend = point(pstart + ptest);
 break;
 */


 int i6;
 int j6;
 for(i6 = (int) p.getX()+1;i6%coarse!=0;i6++){}
 for(j6 = (int) p.getZ();j6%coarse!=0;j6--){}

 pstart = point((float) i6,p.getY(),(float) j6);
 pend = point(pstart + (ptest*coarse));
 break;

case 6:
```

```
   /*
   pstart = point((float) ((int) p.getX()),(float) ((int) p.getY()),(float) ((int) p.getZ()));
   pend = point(pstart +ptest);
   break;
   */

   int i7,j7,k7;

   for(i7 = (int) p.getX();i7%coarse!=0;i7--){}
   for(j7 = (int) p.getY();j7%coarse!=0;j7--){}
   for(k7 = (int) p.getZ();k7%coarse!=0;k7--){}

   pstart = point((float) i7,(float) j7,(float) k7);
   pend = point(pstart +(ptest*coarse));
   break;

  }

 setp1(pstart);
 setp2(pend);
 direction = dir;
}



void SimplexEdge::operator=(const SimplexEdge & se){
 setp1(se.getp1());
 setp2(se.getp2());
 direction = se.getDirection();
}


line_segment SimplexEdge::getLineSegment() const{
 return line_segment(getp1(),getp2());
}

float SimplexEdge::distanceFromP(const point & ptest, int pointnumber){
 int direction = getDirection();
 float distance;
 switch(direction){

 case 0:
  if (pointnumber == 1){
   distance = ptest.getX() - (getp1().getX());
  }
  else{
   distance = (getp2().getX()) - ptest.getX();
  }
  break;

 case 1:
  if (pointnumber ==1){
   distance = ptest.getY() - (getp1().getY());
  }
```

```
  else{
    distance = (getp2().getY()) - ptest.getY();
  }
  break;

case 2:
  if (pointnumber ==1){
    distance = ptest.getZ() - (getp1().getZ());
  }
  else{
    distance = (getp2().getZ()) - ptest.getZ();
  }
  break;

case 3:
  if (pointnumber ==1){
    distance = ptest.getX() - (getp1().getX());
  }
  else{
    distance = (getp2().getX()) - ptest.getX();
  }
  break;

case 4:
   if (pointnumber ==1){
    distance = ptest.getY() - (getp1().getY());
  }
  else{
    distance = (getp2().getY()) - ptest.getY();
  }
  break;

case 5:

  if (pointnumber ==1){
    distance = ptest.getZ() - (getp1().getZ());
  }
  else{
    distance = (getp2().getZ()) - ptest.getZ();
  }
  break;

case 6:

  if (pointnumber ==1){
    distance = ptest.getX() - (getp1().getX());
  }
  else{
    distance = (getp2().getX()) - ptest.getX();
  }
  break;
 }
 return distance;
}
```

```
/*=========================================================================*/
/*Simplicial Grid class holds the information about the dimensions of  */
/*the grid and also holds a table  that relate simplicies to eachother  */
/*and tables that relate simplicies to their bounding verticies. .    */

/*************************************************************************/
/*************************************************************************/
/*************************************************************************/



/*************************************************************************/
/*************************************************************************/
/*************************************************************************/

struct SimplexTypePointPair{
 int c;
 point pointlist[4];
};


struct EdgeSLPair{
 int NumberIncident;
 Simplex simplexlist[6];
};


/*************************************************************************/
/*************************************************************************/
/*************************************************************************/

class SimplicialGrid{
 friend class EtoSIterator;
 friend class StoPIterator;
 int xsize;
 int ysize;
 int zsize;
 int coarseness;



 struct SimplexTypePointPair TypetoPoint[6];
 struct EdgeSLPair EdgetoSimplicies[7];

 void initEtoS();
 void initTtoP();

public:

 SimplicialGrid(int,int,int/*dimensions*/,int /*coarseness*/);

 class OutofBoundsException{};
```

```
 int getXSize() const;
 int getYSize() const;
 int getZSize() const;
 int getCoarse() const;

};


int SimplicialGrid::getXSize() const{
 return xsize;
}


int SimplicialGrid::getYSize() const{
 return ysize;
}


int SimplicialGrid::getZSize() const{
 return zsize;
}

int SimplicialGrid::getCoarse() const{
 return coarseness;
}


void SimplicialGrid::initEtoS(){

 EdgetoSimplicies[0].NumberIncident = 6;
 EdgetoSimplicies[0].simplexlist[0] = Simplex(point(0,0,0),3);
 EdgetoSimplicies[0].simplexlist[1] = Simplex(point(0,0,0),4);
 EdgetoSimplicies[0].simplexlist[2] = Simplex(point(0,-1*getCoarse(),0),0);
 EdgetoSimplicies[0].simplexlist[3] = Simplex(point(0,0,-1*getCoarse()),5);
 EdgetoSimplicies[0].simplexlist[4] = Simplex(point(0,-1*getCoarse(),-1*getCoarse()),1);
 EdgetoSimplicies[0].simplexlist[5] = Simplex(point(0,-1*getCoarse(),-1*getCoarse()),2);

 EdgetoSimplicies[1].NumberIncident = 4;
  EdgetoSimplicies[1].simplexlist[0] = Simplex(point(0,0,0),0);
 EdgetoSimplicies[1].simplexlist[1] = Simplex(point(0,0,-1*getCoarse()),1);
 EdgetoSimplicies[1].simplexlist[2] = Simplex(point(-1*getCoarse(),0,0),3);
 EdgetoSimplicies[1].simplexlist[3] = Simplex(point(-1*getCoarse(),0,-1*getCoarse()),5);

 EdgetoSimplicies[2].NumberIncident = 6;
 EdgetoSimplicies[2].simplexlist[0] = Simplex(point(0,0,0),1);
 EdgetoSimplicies[2].simplexlist[1] = Simplex(point(0,0,0),4);
 EdgetoSimplicies[2].simplexlist[2] = Simplex(point(0,-1*getCoarse(),0),0);
 EdgetoSimplicies[2].simplexlist[3] = Simplex(point(-1*getCoarse(),-1*getCoarse(),0),2);
 EdgetoSimplicies[2].simplexlist[4] = Simplex(point(-1*getCoarse(),-1*getCoarse(),0),3);
 EdgetoSimplicies[2].simplexlist[5] = Simplex(point(-1*getCoarse(),0,0),5);

 EdgetoSimplicies[3].NumberIncident = 6;
 EdgetoSimplicies[3].simplexlist[0] = Simplex(point(0,0,0),0);
 EdgetoSimplicies[3].simplexlist[1] = Simplex(point(0,0,0),2);
```

```
 EdgetoSimplicies[3].simplexlist[2] = Simplex(point(0,0,0),3);
 EdgetoSimplicies[3].simplexlist[3] = Simplex(point(0,0,-1*getCoarse()),1);
 EdgetoSimplicies[3].simplexlist[4] = Simplex(point(0,0,-1*getCoarse()),4);
 EdgetoSimplicies[3].simplexlist[5] = Simplex(point(0,0,-1*getCoarse()),5);

 EdgetoSimplicies[4].NumberIncident = 6;
 EdgetoSimplicies[4].simplexlist[0] = Simplex(point(0,0,0),0);
 EdgetoSimplicies[4].simplexlist[1] = Simplex(point(0,0,0),1);
 EdgetoSimplicies[4].simplexlist[2] = Simplex(point(0,0,0),2);
 EdgetoSimplicies[4].simplexlist[3] = Simplex(point(-1*getCoarse(),0,0),3);
 EdgetoSimplicies[4].simplexlist[4] = Simplex(point(-1*getCoarse(),0,0),4);
 EdgetoSimplicies[4].simplexlist[5] = Simplex(point(-1*getCoarse(),0,0),5);

 EdgetoSimplicies[5].NumberIncident = 4;
 EdgetoSimplicies[5].simplexlist[0] = Simplex(point(-1*getCoarse(),0,0),4);
 EdgetoSimplicies[5].simplexlist[1] = Simplex(point(-1*getCoarse(),0,0),5);
 EdgetoSimplicies[5].simplexlist[2] = Simplex(point(-1*getCoarse(),-1*getCoarse(),0),0);
 EdgetoSimplicies[5].simplexlist[3] = Simplex(point(-1*getCoarse(),-1*getCoarse(),0),2);

 EdgetoSimplicies[6].NumberIncident = 4;
 EdgetoSimplicies[6].simplexlist[0] = Simplex(point(0,0,0),1);
 EdgetoSimplicies[6].simplexlist[1] = Simplex(point(0,0,0),2);
 EdgetoSimplicies[6].simplexlist[2] = Simplex(point(0,0,0),3);
 EdgetoSimplicies[6].simplexlist[3] = Simplex(point(0,0,0),4);

}



void SimplicialGrid::initTtoP(){

 TypetoPoint[0].c = 0;
 TypetoPoint[0].pointlist[0] = point(0,0,0);
 TypetoPoint[0].pointlist[1] = point(0,1*getCoarse(),0);
 TypetoPoint[0].pointlist[2] = point(1*getCoarse(),1*getCoarse(),0);
 TypetoPoint[0].pointlist[3] = point(0,1*getCoarse(),1*getCoarse());

 TypetoPoint[1].c = 1;
 TypetoPoint[1].pointlist[0] = point(0,0,0);
 TypetoPoint[1].pointlist[1] = point(0,0,1*getCoarse());
 TypetoPoint[1].pointlist[2] = point(1*getCoarse(),1*getCoarse(),1*getCoarse());
 TypetoPoint[1].pointlist[3] = point(0,1*getCoarse(),1*getCoarse());

 TypetoPoint[2].c = 2;
 TypetoPoint[2].pointlist[0] = point(0,0,0);
 TypetoPoint[2].pointlist[1] = point(1*getCoarse(),1*getCoarse(),1*getCoarse());
 TypetoPoint[2].pointlist[2] = point(1*getCoarse(),1*getCoarse(),0);
 TypetoPoint[2].pointlist[3] = point(0,1*getCoarse(),1*getCoarse());

 TypetoPoint[3].c = 3;
 TypetoPoint[3].pointlist[0] = point(0,0,0);
 TypetoPoint[3].pointlist[1] = point(1*getCoarse(),0,0);
 TypetoPoint[3].pointlist[2] = point(1*getCoarse(),1*getCoarse(),0);
 TypetoPoint[3].pointlist[3] = point(1*getCoarse(),1*getCoarse(),1*getCoarse());
```

```
 TypetoPoint[4].c = 4;
 TypetoPoint[4].pointlist[0] = point(0,0,0);
 TypetoPoint[4].pointlist[1] = point(0,0,1*getCoarse());
 TypetoPoint[4].pointlist[2] = point(1*getCoarse(),0,0);
 TypetoPoint[4].pointlist[3] = point(1*getCoarse(),1*getCoarse(),1*getCoarse());

 TypetoPoint[5].c = 5;
 TypetoPoint[5].pointlist[0] = point(1*getCoarse(),0,0);
 TypetoPoint[5].pointlist[1] = point(0,0,1*getCoarse());
 TypetoPoint[5].pointlist[2] = point(1*getCoarse(),1*getCoarse(),1*getCoarse());
 TypetoPoint[5].pointlist[3] = point(1*getCoarse(),0,1*getCoarse());
}


SimplicialGrid::SimplicialGrid(int x,int y,int z, int coarse){
 xsize =x;
 ysize =y;
 zsize =z;
 coarseness = coarse;
 initTtoP();
 initEtoS();
}


class EtoSIterator{

 int Direction;
 point Location;
 int HowManyLeft;

public:

 EtoSIterator(const SimplexEdge&, const SimplicialGrid&);
 Simplex nextSimplex(const SimplicialGrid&);
 class NoSimpliciesLeftException{};

};


EtoSIterator::EtoSIterator(const SimplexEdge &e, const SimplicialGrid &g){
 Direction = e.getDirection();
 Location = e.getp1();
 HowManyLeft = g.EdgetoSimplicies[Direction].NumberIncident;
}

Simplex EtoSIterator::nextSimplex(const SimplicialGrid &g){
 if (HowManyLeft == 0) throw NoSimpliciesLeftException();
 else{
   Simplex s(Location + (g.EdgetoSimplicies[Direction].simplexlist[HowManyLeft -1]).getLocation(),(g.Edgeto-
Simplicies[Direction].simplexlist[HowManyLeft -1]).getType());
   HowManyLeft--;
   return s;
 }
```

```
}

//*********************************************************************/
//*********************************************************************/
//*********************************************************************/

class StoPIterator{

 int TypeOfSimplex;
 point Location;
 int HowManyLeft;

public:

 StoPIterator(const Simplex &);
 point nextPoint(const SimplicialGrid &);
 class NoPointsLeftException{};
};

StoPIterator::StoPIterator(const Simplex & s){
 TypeOfSimplex = s.getType();
 Location = s.getLocation();
 HowManyLeft =4;
}

point StoPIterator::nextPoint(const SimplicialGrid &g){
 if (HowManyLeft ==0) throw NoPointsLeftException();
 else{

  point p(Location + g.TypetoPoint[TypeOfSimplex].pointlist[HowManyLeft-1]);
  HowManyLeft --;
  return p;
 }
}




class SimplexEdgePointPair{

 SimplexEdge s;
 point p;
 void sets(const SimplexEdge &);
 void setp(const point &);

public:
 SimplexEdgePointPair();
 SimplexEdgePointPair(const point &,const SimplexEdge &);
 SimplexEdgePointPair(const SimplexEdgePointPair &);
 point getPoint() const;
 SimplexEdge getSimplexEdge() const;
};
```

```
SimplexEdgePointPair::SimplexEdgePointPair(){
 s = SimplexEdge();
 p = point();
}

void SimplexEdgePointPair::sets(const SimplexEdge &se){
s = se;

}

void SimplexEdgePointPair::setp(const point &po){
p=po;

}

point SimplexEdgePointPair::getPoint() const{
 return p;
}

SimplexEdge SimplexEdgePointPair::getSimplexEdge() const{
 return s;
}

SimplexEdgePointPair::SimplexEdgePointPair(const SimplexEdgePointPair &spp){
 sets(spp.getSimplexEdge());
 setp(spp.getPoint());
}

SimplexEdgePointPair::SimplexEdgePointPair(const point & po, const SimplexEdge & se){

 sets(se);
 setp(po);
}

class comparesimplex{
public:
 bool operator()(Simplex,Simplex);
};


bool comparesimplex::operator()(Simplex a,Simplex b){

 if(a.getLocation()<b.getLocation())
  return true;
 else{

  if (a.getLocation() == b.getLocation())
    {
     if (a.getType()<b.getType()) return true;
    }
 }

 return false;
```

```
}


ostream& operator<<(ostream& s,Simplex &Si){
 //Stream output operator for simplicies.

 return s<<"[Location: "<<Si.getLocation()<<" Type: "<<Si.getType()<<"]";
}

ostream& operator<<(ostream& s,SimplexEdge &se){
 //Stream output operator for simplexedges.

 return s<<"[p1: "<< se.getp1() <<" p2: "<<se.getp2()<<" Direction: "
      << se.getDirection()<<"]";
}


ostream& operator<<(ostream& s,SimplexEdgePointPair &Sip){
 //Stream output operator for simplexedgepointpairs.

 return s<<"[SimplexEdge: "<<Sip.getSimplexEdge()
      <<" Point: "<<Sip.getPoint()<<"]";
}
```

Appendix E

Topological Transformation Library and Source Code

E.1 Header File (TTransformer.h)

```
/*** 3-D Topological Transformer... To be used with Segmentation Algorithm****/
/*** Author: Varouj A. Chitilian****/
/*** Date: Dec 15, 1998***/
/*** Email: vchitil@mit.edu,vchitilian@sarnoff.com**/
/*** The function that reparameterizes tosses the reparameterized points**/
/*** into a list that contains unit cells and corresponding points***/
/*** This data structure is used to perform the topological transformations***/
/*** and will therefore be called a topological transformer.**/

/***The topological transformer will also maintain a list that, specifies */
/***whether given simplex verticies are in or out of the surface at any */
/***given point in time...                                              */



/*** The containers of the transformer will look like:                  */


/*** Unit Boundary Cell ========> list of points                        ***/
/*** Unit Boundary Cell ========> list of points                        ***/
/*** Unit Boundary Cell ========> list of points                        ***/
/***     .                .                    ***/
/***     .                .                    ***/
/***     .                .                    ***/


/** Simplex Vertex ======> in/out (boolean) ***/
/** Simplex Vertex ======> in/out (boolean) ***/
/** Simplex Vertex ======> in/out (boolean) ***/
/**     .              .      ***/
/**     .              .      ***/
/**     .              .      ***/
/**     .              .      ***/



/*** the functions of the transformer will include:                     */
/*** - construction of transformer container using Surface, which includes: */
/***     - finding intersection between unit cell edges and planes      */
/***     - ray casting to see which vertices are in and which are out   */
/*** - transformation: which takes the transformer and forms a surface  */
```

```
#include <stdlib.h>
#include <iostream.h>
#include "3dmodel.h"
#include "3dsimplex.h"



class NotBoundaryException{};
class ThreePlusException{};
class TwoPlusException{};


class TopologicalTransformer{

  map<Simplex,list<SimplexEdgePointPair>,comparesimplex> RPoints;
  map<point,bool,comparepoints> IsIn;
    void isBoundaryCell(Simplex, const SimplicialGrid &) throw (NotBoundaryException,ThreePlusExcep-
tion,TwoPlusException);
  bool specialEdge(const SimplexEdge &);
  void insertIntoRPoints(const Simplex&,SimplexEdgePointPair);
  plane_segment threePlusFindPlane(Simplex ,list<SimplexEdgePointPair>);
  void twoPlusFindPlanes(const Simplex &,list<SimplexEdgePointPair>, list<plane_segment> &);

public:

  TopologicalTransformer(const Surface&, const SimplicialGrid &);
  void Transform(Surface &, const SimplicialGrid &);

};
```

E.2 Source Code (TTransformer.cpp)

```
/*** 3-D Topological Transformer... To be used with Segmentation Algorithm****/
/*** Author: Varouj A. Chitilian****/
/*** Date: Dec 15, 1998***/
/*** Email: vchitil@mit.edu,vchitilian@sarnoff.com**/
/*** The function that reparameterizes tosses the reparameterized points**/
/*** into a list that contains unit cells and corresponding points***/
/*** This data structure is used to perform the topological transformations***/
/*** and will therefore be called a topological transformer.**/

/***The topological transformer will also maintain a list that, specifies */
/***whether given simplex verticies are in or out of the surface at any */
/***given point in time...                                              */
```

```
/*** The containers of the transformer will look like:          */


/*** Unit Boundary Cell ========> list of points                 ***/
/*** Unit Boundary Cell ========> list of points                 ***/
/*** Unit Boundary Cell ========> list of points                 ***/
/***     .                .                ***/
/***     .                .                ***/
/***     .                .                ***/


/** Simplex Vertex =======> in/out (boolean) ***/
/** Simplex Vertex =======> in/out (boolean) ***/
/** Simplex Vertex =======> in/out (boolean) ***/
/**     .                .       ***/
/**     .                .       ***/
/**     .                .       ***/
/**     .                .       ***/



/*** the functions of the transformer will include:             */
/*** - construction of transformer container using Surface, which includes: */
/***     - finding intersection between unit cell edges and planes     */
/***     - ray casting to see which vertices are in and which are out   */
/*** - transformation: which takes the transformer and forms a surface    */



#include <stdlib.h>
#include <iostream.h>
#include "3dmodel.h"
#include "3dsimplex.h"



class NotBoundaryException{};
class ThreePlusException{};
class TwoPlusException{};


class TopologicalTransformer{


  map<Simplex,list<SimplexEdgePointPair>,comparesimplex> RPoints;
  map<point,bool,comparepoints> IsIn;
    void isBoundaryCell(Simplex, const SimplicialGrid &) throw (NotBoundaryException,ThreePlusException,TwoPlusException);
  bool specialEdge(const SimplexEdge &);
  void insertIntoRPoints(Simplex ,SimplexEdgePointPair);
  plane_segment threePlusFindPlane(const Simplex &,list<SimplexEdgePointPair>);
```

```
 void twoPlusFindPlanes(const Simplex &,list<SimplexEdgePointPair>, list<plane_segment> &);

public:

 TopologicalTransformer(const Surface&, const SimplicialGrid &);
 void Transform(Surface &, const SimplicialGrid &);

};



void TopologicalTransformer::insertIntoRPoints( Simplex s,SimplexEdgePointPair se){

  (RPoints[s]).push_back(se);
}



//Performs Reparameterization!!!!

TopologicalTransformer::TopologicalTransformer(const Surface& s, const SimplicialGrid & grid){
 PlaneIterator PI(s);

 map<point,list<float>,comparepoints> IsInHolder;
 //cout<< "Please Work";
 try{
  while(1){
   plane_segment  testplane(PI.getNextPlane());
   for (int direction=0;direction<7;direction++){
    list<line_segment> linelist;
    testplane.whichSimplexLinesIntersect(linelist,direction,grid.getCoarse());
    list<line_segment>::iterator iter;
    for(iter = linelist.begin();iter!= linelist.end();iter++){
     point p(testplane.intersectsLine(*iter));

     //Getting RayCasting information as we reparameterize
     if (direction == 0){
      point p2(p.getY(),p.getZ(),0);
      (IsInHolder[p2]).push_back(p.getX());
      (IsInHolder[p2]).sort();
     }

     SimplexEdge g(p,*iter,grid.getCoarse());
     SimplexEdgePointPair pair(p,g);
     EtoSIterator etos(g,grid);
     try{
      while(1){
       insertIntoRPoints(etos.nextSimplex(grid),pair);
      }
     }
     catch (EtoSIterator::NoSimpliciesLeftException){}
    }
```

```
   }
  }
}
catch (PlaneIterator::NoMoreException){}


//print out RPoints...
map<Simplex,list<SimplexEdgePointPair>,comparesimplex>::iterator RPointsiter;
list<SimplexEdgePointPair>::iterator liter;

cout<<"\n\nPRINTING OUT RPOINTS\n\n";

for(RPointsiter = RPoints.begin();RPointsiter!=RPoints.end();RPointsiter++){
 cout<< "\n\nSimplex: "<<(*RPointsiter).first<<" List: ";
 for(liter = (*RPointsiter).second.begin();
    liter!= (*RPointsiter).second.end();
    liter++){
   cout<< "\n"<<(*liter) <<" ";

 }
}

//RPoints size...

cout<<"\n\nRPOINTS SIZE\n\n";
cout<<RPoints.size()<<"\n\n";

//The rest of the function fills in IsIn data structure

//int size = IsInHolder.size();
int j1;
for(j1 =0;j1<grid.getYSize();j1++){
 int k1;
 for(k1 =0;k1<grid.getZSize();k1++){
    if(IsInHolder.find(point(j1,k1,0))!=IsInHolder.end()){
      int size;
      size = IsInHolder[point(j1,k1,0)].size();
      float holder1;
      holder1 = (IsInHolder[point(j1,k1,0)]).front();
      (IsInHolder[point(j1,k1,0)]).pop_front();
      float holder2;

      int i1;
      for( i1=0;i1<((int) holder1) +1;i1++){
       //these lines not necessary since if entry not in map, false is
       //automatically returned.

       // IsIn[point(i1,j1,k1)] = false;
      }
      int count;
      for(count = 0;count<size-1 ;count++){
       holder2 = (IsInHolder[point(j1,k1,0)]).front();
       (IsInHolder[point(j1,k1,0)]).pop_front();
```

```
        int aeta;
        for(aeta= ((int) holder1)+1;aeta< ((int) holder2)+1;aeta++){
         if(count%2 ==0){
      IsIn[point(aeta,j1,k1)] =  true;
          }
         else
      if (count%2 ==1){
       //IsIn[point(aeta,j1,k1)] = false;
      }
        }

        holder1 = holder2;
       }

       //this will be the last interval

       int i5;
       for (i5 = ((int) holder1) +1 ; i5 <grid.getXSize();i5++){
        //IsIn[point(i5,j1,k1)] = false;
       }
      }//if found in isinholder
      else{
       //the whole row is false
       int i6;
       for (i6 =0;i6<grid.getXSize();i6++){
        //IsIn[point(i6,j1,k1)] = false;
       }
      }

    }//k1
  }//j1

  //print out ISIn data stucture

  map<point,bool,comparepoints>::iterator IsIniter;

  cout<<"\n\nPRINTING OUT IsIn\n\n";

  for(IsIniter = IsIn.begin();IsIniter!=IsIn.end();IsIniter++){
   cout<< "\nPoint: "<<(*IsIniter).first<<" IsIn? "
      << (*IsIniter).second ;
  }



}
```

```
void TopologicalTransformer::isBoundaryCell(Simplex s,const SimplicialGrid & g)
throw (NotBoundaryException, ThreePlusException,TwoPlusException){
 int NumberIn;
 int NumberOut;

 NumberIn = 0;
 NumberOut = 0;

 StoPIterator Iter(s);

 while(1){
  try{

    point p(Iter.nextPoint(g));
    if (IsIn[p] == true)
      NumberIn++;
    else NumberOut++;
   }

   catch (StoPIterator::NoPointsLeftException){
    break;
   }
 }

 if (NumberIn==4 || NumberOut ==4) throw NotBoundaryException();
 if (NumberIn==3 || NumberOut ==3) throw ThreePlusException();
 if (NumberIn ==2) throw TwoPlusException();
}



bool TopologicalTransformer::specialEdge(const SimplexEdge & se){

 bool P1In;

 bool P2In;

 P1In = false;
 P2In = false;

 point p1(se.getp1());
 point p2(se.getp2());

 if (IsIn[p1]) P1In = true;
 if (IsIn[p2]) P2In = true;

 if ((P1In && !P2In) || (!P1In && P2In)) return true;
 return false;
}


plane_segment TopologicalTransformer::threePlusFindPlane(const Simplex &s,
```

```
    list<SimplexEdgePointPair> sepp){

int sz = sepp.size();

if(sz<3){
 cout<<"We have problems in threePlusFindPlane().... exiting";
 exit(0);
}

if(sz ==3){
 point pointlist[3];
 list<SimplexEdgePointPair>::iterator iter;
 int i;
 for(i =0,iter = sepp.begin();iter!= sepp.end();i++,iter++){
  pointlist[i] = (*iter).getPoint();
 }
 return plane_segment(pointlist[0],pointlist[1],pointlist[2]);

}


//This is complicated because we have to choose points not only that are
//on specialEdges but also if there are >1 points on those special Edges
//we have to choose those that are closer to the outside point of the
//Special edge in question
//This case is the general case where a Topological Transformation is
//needed.

if(sz>3){

 int addflag = 1;


 point pointlist[3];
 list<SimplexEdgePointPair> sep;
 list<float> mindists;

 list<SimplexEdgePointPair>::iterator iter1,iter2;
 list<float>::iterator distiter;
 //first we need an initial item in sep and mindists so that the
 //two for loops will work properly together

 for(iter1 = sepp.begin();sep.size()==0;iter1++){

  SimplexEdge setest((*iter1).getSimplexEdge());
  point ptest((*iter1).getPoint());
  if(specialEdge(setest)){
   sep.push_back(*iter1);

   //we want the closest point to the outside point of the special edge

   if(IsIn[setest.getp2()])
    mindists.push_back(setest.distanceFromP(ptest,0));
   else mindists.push_back(setest.distanceFromP(ptest,1));
```

```
  }
}



for(;iter1!=sepp.end();iter1++){

 if (specialEdge((*iter1).getSimplexEdge())){

   addflag =1;
   SimplexEdge setest1((*iter1).getSimplexEdge());
   point ptest1((*iter1).getPoint());

   float dtest;
   if(IsIn[setest1.getp2()])
    dtest=setest1.distanceFromP(ptest1,0);
   else dtest = setest1.distanceFromP(ptest1,1);


   for(iter2 = sep.begin(),distiter = mindists.begin();
      iter2!= sep.end(); distiter++,iter2++){

     SimplexEdge setest2((*iter2).getSimplexEdge());

     if(setest1.getDirection() == setest2.getDirection()){

       if((*distiter)>dtest){
         sep.erase(iter2);
         mindists.erase(distiter);
         sep.push_back(*iter1);
         mindists.push_back(dtest);
       }

       addflag = 0;
       break;
     }


   }
   if (addflag){
     sep.push_back(*iter1);
     mindists.push_back(dtest);
   }

 }
}

int j;
for (j=0,iter1=sep.begin();iter1!=sep.end();j++,iter1++){
 if (j>2){
   cout<<"We gots problems in threeplusfindplane >3... exiting";
   exit(0);
 }
```

```
      pointlist[j] = point((*iter1).getPoint());
     }

     return plane_segment(pointlist[0],pointlist[1],pointlist[2]);

    }

   }

  void TopologicalTransformer::twoPlusFindPlanes(const Simplex &s, list<SimplexEdgePointPair> sepp,
list<plane_segment> &lp){
   //Whenever we have a twoPlusFindPlanes called we expect to have four relevant
   //points on the Simplex, and therefore must return two planes. This is how
   //the three plus and two plus differ. The three plus returns just a single
   //plane while the two plus takes a reference to a list of planes and places
   //two planes in the list. Besides the returning of the planes much of the
   //structure of this function is the same as the ThreePlus case.

   int sz = sepp.size();

   if(sz<4){
    list<SimplexEdgePointPair>::iterator iterz;
    for(iterz = sepp.begin();iterz!= sepp.end();iterz++){
     cout<<"\n\nSimplexEdge: "<<(*iterz).getSimplexEdge();
     cout<<"\nPoint: "<<(*iterz).getPoint();
     cout<<"\nThe Simplex is : "<<s;
     cout<<"\nLaterz we're done...\n\n";
    }
    cout<<"We have problems in TwoPlusFindPlane().... exiting";
    exit(0);
   }


   //figure out a way to find which points to connect using concept of
   //special edge


   if(sz ==4){

    SimplexEdgePointPair sepoints[4];

    list<SimplexEdgePointPair>::iterator iter;
    int i;
    for(i =0,iter = sepp.begin();iter!= sepp.end();i++,iter++){
     sepoints[i] = (*iter);
    }

    plane_segment plane1((sepoints[0]).getPoint(),(sepoints[1]).getPoint(),
       (sepoints[2]).getPoint());

    point newpoints[3];
    newpoints[0] = (sepoints[3]).getPoint();
```

```
SimplexEdge se0((sepoints[0]).getSimplexEdge());
SimplexEdge se1((sepoints[1]).getSimplexEdge());
SimplexEdge se2((sepoints[2]).getSimplexEdge());
SimplexEdge se3((sepoints[3]).getSimplexEdge());

int edgenumber =0;
int number =1;

if(se3.getp1() == se0.getp1() || se3.getp1() == se0.getp2() ||
  se3.getp2() == se0.getp1() || se3.getp2() == se0.getp2()){
 newpoints[number] = (sepoints[edgenumber]).getPoint();
 number++;
}
edgenumber++;

if(se3.getp1() == se1.getp1() || se3.getp1() == se1.getp2() ||
  se3.getp2() == se1.getp1() || se3.getp2() == se1.getp2()){
 newpoints[number] = (sepoints[edgenumber]).getPoint();
 number++;
}
edgenumber++;

if(se3.getp1() == se2.getp1() || se3.getp1() == se2.getp2() ||
  se3.getp2() == se2.getp1() || se3.getp2() == se2.getp2()){
 newpoints[number] = (sepoints[edgenumber]).getPoint();
 number++;
}

if (number != 3){
 cout<< "problems in twooutfindplane... exiting";
 exit(0);
}

plane_segment plane2(newpoints[0],newpoints[1],newpoints[2]);

lp.push_back(plane1);
lp.push_back(plane2);
}


//This is complicated because we have to choose points not only that are
//on specialEdges but also if there are >1 points on those special Edges
//we have to choose those that are closer to the outside point of the
//Special edge in question
//This case is the general case where a Topological Transformation is
//needed.

if(sz>4){

 int addflag = 1;



 list<SimplexEdgePointPair> sep;
```

```
list<float> mindists;

list<SimplexEdgePointPair>::iterator iter1,iter2;
list<float>::iterator distiter;
//first we need an initial item in sep and mindists so that the
//two for loops will work properly together

for(iter1 = sepp.begin();sep.size()==0;iter1++){

 SimplexEdge setest((*iter1).getSimplexEdge());
 point ptest((*iter1).getPoint());
 if(specialEdge(setest)){
  sep.push_back(*iter1);

  //we want the closest point to the outside point of the special edge

  if(IsIn[setest.getp2()])
   mindists.push_back(setest.distanceFromP(ptest,0));
  else mindists.push_back(setest.distanceFromP(ptest,1));

 }
}



for(;iter1!=sepp.end();iter1++){

 if (specialEdge((*iter1).getSimplexEdge())){

  addflag =1;
  SimplexEdge setest1((*iter1).getSimplexEdge());
  point ptest1((*iter1).getPoint());

  float dtest;
  if(IsIn[setest1.getp2()])
   dtest=setest1.distanceFromP(ptest1,0);
  else dtest = setest1.distanceFromP(ptest1,1);


  for(iter2 = sep.begin(),distiter = mindists.begin();
    iter2!= sep.end(); distiter++,iter2++){

   SimplexEdge setest2((*iter2).getSimplexEdge());

   if(setest1.getDirection() == setest2.getDirection()){

    if((*distiter)>dtest){
     sep.erase(iter2);
     mindists.erase(distiter);
     sep.push_back(*iter1);
     mindists.push_back(dtest);
    }

    addflag = 0;
```

```
      break;
     }


   }
   if (addflag){
    sep.push_back(*iter1);
    mindists.push_back(dtest);
   }

 }
}

//make sure there's only four things in sep.

if (sep.size()!= 4){
 cout<<"sep has more than 4 entries... exiting";
 exit(0);
}

//now take sep and transform it into two planes

list<SimplexEdgePointPair>::iterator final;
SimplexEdgePointPair sepoints[4];

int n;
for(n=0,final = sep.begin();final != sep.end();n++,final++)
 sepoints[n] = (*final);

plane_segment plane1((sepoints[0]).getPoint(),(sepoints[1]).getPoint(),
    (sepoints[2]).getPoint());

point newpoints[3];
newpoints[0] = (sepoints[3]).getPoint();

SimplexEdge se0((sepoints[0]).getSimplexEdge());
SimplexEdge se1((sepoints[1]).getSimplexEdge());
SimplexEdge se2((sepoints[2]).getSimplexEdge());
SimplexEdge se3((sepoints[3]).getSimplexEdge());

int edgenumber =0;
int number =1;

if(se3.getp1() == se0.getp1() || se3.getp1() == se0.getp2() ||
  se3.getp2() == se0.getp1() || se3.getp2() == se0.getp2()){
 newpoints[number] = (sepoints[edgenumber]).getPoint();
 number++;
}
edgenumber++;

if(se3.getp1() == se1.getp1() || se3.getp1() == se1.getp2() ||
  se3.getp2() == se1.getp1() || se3.getp2() == se1.getp2()){
 newpoints[number] = (sepoints[edgenumber]).getPoint();
 number++;
```

```
    }
    edgenumber++;

    if(se3.getp1() == se2.getp1() || se3.getp1() == se2.getp2() ||
      se3.getp2() == se2.getp1() || se3.getp2() == se2.getp2()){
     newpoints[number] = (sepoints[edgenumber]).getPoint();
     number++;
    }

    if (number != 3){
     cout<< "problems in twooutfindplane... exiting";
     exit(0);
    }

    plane_segment plane2(newpoints[0],newpoints[1],newpoints[2]);

    lp.push_back(plane1);
    lp.push_back(plane2);
   }

}
```

```
void TopologicalTransformer::Transform(Surface & newSurface, const SimplicialGrid &g){

 //print out newSurface for debugging purposes...

 newSurface.setIsIn(IsIn);
 AdjacencyList::iterator yo1;
 /*
 cout<<"\n\nThe NEW SURFACE 1\n\n";

 for(yo1 = newSurface.begin();yo1!= newSurface.end();yo1++){

  cout<<"\n"<<(*yo1).first;
 }
 */

 map<Simplex,list<SimplexEdgePointPair>,comparesimplex>::iterator iter;
 for(iter = RPoints.begin();iter!=RPoints.end();iter++){
  try{
   isBoundaryCell((*iter).first,g);
  }
  catch (ThreePlusException){

   newSurface.insert(threePlusFindPlane((*iter).first,(*iter).second));
   newSurface.incNumPlanes();
  }
```

```
  catch (TwoPlusException){
    list<plane_segment> planelist;
    twoPlusFindPlanes((*iter).first,(*iter).second, planelist);
    list<plane_segment>::iterator iter3;
    for(iter3 = planelist.begin();iter3 != planelist.end();iter3++){
      newSurface.insert(*iter3);
      newSurface.incNumPlanes();
    }
  }
  catch (NotBoundaryException){}
 }
 //Print out the NewSurface for Debugging Purposes

 AdjacencyList::iterator yo;
 /*cout<<"\n\nThe NEW SURFACE shit\n\n";

 for(yo = newSurface.begin();yo!= newSurface.end();yo++){

  cout<<"\n"<<(*yo).first;
 }
 */

}
```

Appendix F

Segmentation Library and Source Code

F.1 Header File (Segmentor.h)

```
//Segmentor.h

#include<stdlib.h>
#include<iostream.h>
#include<map.h>
#include "TTransformer.h"



class Segmentor{

 float mincontourspeed;
 Surface OptimalSurface;
 void setOptimalSurface(Surface);

public:

 map<point,bool,comparepoints> burned;
 Segmentor(Surface);
 Surface getOptimalSurface();
 void updateMin(Surface);
 void updateBurned(map<point,bool,comparepoints> &);
 float getMinSpeed();
 void setMinSpeed(float s);
};
```

F.2 Source Code (Segmentor.cpp)

```
#include<stdlib.h>
#include<iostream.h>
#include<map.h>
#include "TTransformer.h"



class Segmentor{
```

```
 float mincontourspeed;
 Surface OptimalSurface;
 void setOptimalSurface(Surface);
 int iterations;

public:

 map<point,bool,comparepoints> burned;
 Segmentor(Surface);
 Surface getOptimalSurface();
 void updateMin(Surface);
 void updateBurned(map<point,bool,comparepoints> &);
 float getMinSpeed();
 void setMinSpeed(float);
};



Segmentor::Segmentor(Surface s){

 iterations =0;

 mincontourspeed = s.getContourSpeed();

 OptimalSurface = s;

 burned = s.getIsIn();

}

void Segmentor::setMinSpeed(float s){
 s = mincontourspeed;
}

void Segmentor::updateBurned(map<point,bool,comparepoints> & IsIn){

 map<point,bool,comparepoints>::iterator iter;

 for(iter = IsIn.begin();iter != IsIn.end();iter++){
  burned[(*iter).first] = true;
 }
}


Surface Segmentor::getOptimalSurface(){
 return OptimalSurface;
}

float Segmentor::getMinSpeed(){
 return mincontourspeed;
}
```

```
void Segmentor::setOptimalSurface(Surface s){
 OptimalSurface = s;
}


void Segmentor::updateMin(Surface s){

 if (iterations == 0){
  mincontourspeed = s.getContourSpeed();
  setOptimalSurface(s);
 }
 else{
  if(s.getContourSpeed() < getMinSpeed()){
   mincontourspeed = s.getContourSpeed();
   setOptimalSurface(s);
  }
 }
iterations++;
}
```

Appendix G

Main Program

G.1 Source Code (GeomExtract.cpp)

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include "Segmentor.h"
#include "Image.cpp"


void Transformthesurface(Surface & NewContour, Surface & OldContour, SimplicialGrid & grid){

 TopologicalTransformer Trans(OldContour,grid);

 Trans.Transform(NewContour,grid);

}



main(){


 // cout<<"Don't make any sense";


 SimplicialGrid Grid2(30,30,30,1);
 SimplicialGrid Grid(30,30,30,2);
 Surface Contour;
 /*
  point p1(10.543,11.26,3.55);
 point p2(28.6,11.47,3.93);
 point p3(29.42,10.34,20.79);
 point p4(11.43,11.33,21.42);
 point p5(10.21,27.6,3.02);
 point p6(30.31,27.85,2.97);
 point p7(28.29,28.61,20.19);
 point p8(9.56,29.37,21.79);
 */
 /* point p1(10.543,11.26,3.55);
 point p2(18.6,11.47,3.93);
 point p3(19.42,10.34,10.79);
 point p4(11.43,11.33,11.42);
 point p5(10.21,17.6,3.02);
 point p6(20.31,17.85,2.97);
 point p7(18.29,18.61,10.19);
```

```
point p8(9.56,19.37,11.79);

plane_segment plane1(p1,p2,p3);

plane_segment plane2(p1,p4,p3);

plane_segment plane3(p1,p2,p6);

plane_segment plane4(p1,p5,p6);

plane_segment plane5(p1,p4,p8);

plane_segment plane6(p1,p5,p8);

plane_segment plane7(p2,p6,p7);

plane_segment plane8(p2,p3,p7);

plane_segment plane9(p4,p3,p7);

plane_segment plane10(p4,p8,p7);

plane_segment plane11(p5,p6,p7);

plane_segment plane12(p5,p7,p8);

*/
/*
  plane_segment plane1(point(10.543,11.26,3.55),point(15.21,23.47,15.93),
        point(29.42,10.34,2.18));

 plane_segment plane2(point(10.543,11.26,3.55),point(15.21,23.47,15.93),
    point(12.23,22.76,8.03));

 plane_segment plane3(point(15.21,23.47,15.93), point(29.42,10.34,2.18),
        point(12.23,22.76,8.03));

 plane_segment plane4(point(10.543,11.26,3.55),point(29.42,10.34,2.18),
        point(12.23,22.76,8.03));

*/


plane_segment plane1(point(13.643,14.7632,14.55),point(15.21,16.47,15.93),
        point(16.42,13.3412,13.18));

plane_segment plane2(point(13.643,14.7632,14.55),point(15.21,16.47,15.93),
        point(14.23,15.76,16.03));

plane_segment plane3(point(15.21,16.47,15.93), point(16.42,13.3412,13.18),
        point(14.23,15.76,16.03));

plane_segment plane4(point(13.643,14.7632,14.55),point(16.42,13.3412,13.18),
        point(14.23,15.76,16.03));
```

```
//cout<<"Hello????";

Contour.insert(plane1);

Contour.insert(plane2);

Contour.insert(plane3);

Contour.insert(plane4);

/*Contour.insert(plane5);

Contour.insert(plane6);

Contour.insert(plane7);

Contour.insert(plane8);

Contour.insert(plane9);

Contour.insert(plane10);

Contour.insert(plane11);

Contour.insert(plane12);
*/


Surface NewContour2;


Transformthesurface(NewContour2,Contour,Grid2);


Segmentor Seg(NewContour2);

NewContour2.writeVTKModel("vtkout1.vtk");

int i = 2;

Image<char> image("Test.1");

Image<float> I(image.getVolumeGradient());

// I.displaymatrix15();

while (1){
 try{
```

```
   NewContour2.applySpeed(I);
   Surface NewContour;
   if (i>10){
    Transformthesurface(NewContour,NewContour2,Grid);
   }
   else{
    Transformthesurface(NewContour,NewContour2,Grid2);
   }
   Seg.updateMin(NewContour);
   char str[11];
   sprintf(str,"vtkout%d.vtk",i);
   NewContour.writeVTKModel(str);
   NewContour2 = NewContour;
   i++;
  }
  catch (Surface::SurfaceOutOfBoundsException){break;}
}

Seg.getOptimalSurface().writeVTKModel("vtkoutopt.vtk");

// NewContour.applySpeed();
// Surface NewContour3;
// Transformthesurface(NewContour3,NewContour,Grid2);
//NewContour3.writeVTKModel("vtkout3.vtk");

//Surface NewContour;

//Transformthesurface(NewContour,NewContour2,Grid);

//cout<<"\n\nThe New Contour Size Is\nDrum Roll Please....."<<
```

```
//NewContour.size();


//NewContour.writeVTKModel("vtkout.vtk");


//Testing Plane Iterator for Non Connected Surfaces


//cout<<"\n\nPLANES ARE ";

// PlaneIterator PI(NewContour2);

//while(1){
//try{

// cout<<"\n"<<PI.getNextPlane();
//}
//catch(PlaneIterator::NoMoreException){
// break;
//}
//}



// cout<<"\n\n\n\n"<<"PATCHITERATIOR On FIRST POINT\n\n";
/*
AdjacencyList::iterator itera;

itera = NewContour.begin();

PatchIterator PAI(NewContour,(*itera).first);

while(1){
 try{
  cout<< "\n"<<PAI.getNextPlane();
 }
 catch(PatchIterator::NoMoreException){
  break;
 }
}
*/
//cout<<"\n\nNumber of Connected Components is: "<<NewContour2.getConnComps()<<"\n";
//cout<<"\n\nThe New Contour Size Is\nDrum Roll Please....."<<
//NewContour2.size();
// cout<<"\nContour's IsIN size is: "<<NewContour2.IsInSize()<<"\n";
//cout<<"\nNewContour's ISIn size is: "<<NewContour.IsInSize()<<"\n";

}
```

References

[1] Russ, J., "The Image Processing Handbook Second Edition," CRC Press Inc., 1995.

[2] Orel G., Schnall M., Volsi V., Troupin, R., "Suspicious Breast Lesions: MR Imaging with Radiologic Pathologic Correlation," *Radiology*, 190:pp 485-493, 1994.

[3] Singh A., Goldgof D., Terzopoulos D., "Deformable Models in Medical Image Analysis", IEEE Computer Society Press, 1998.

[4] McInerney T., Terzopoulos D., "Medical Image Segmentation Using Topologically Adaptable Surfaces," *Proc. CVRMed*, Gernoble, France, March, 1997.

[5] Malladi R., Sethian J., Vemuri B., "Shape modeling with front propagation: A level set approach". *IEEE Trans. Pattern Analysis and Machine Intelligence,* 17(2): pp 158-175, 1995.

[6] Kass M., Witken A., Terzopoulos D., "Snakes: Active contour models," International Journal of Computer vision, 1(4): pp 321-331, 1998.

[7] Cohen L., "On active contour models and balloons," *CGVIP: Image Understanding,* 53(2):pp 211-218, March 1991.

[8] McInerney T., Terzopoulos D., "Topologically adaptable snakes," Proc. Fifth International Conference on Computer Vision (ICCV '95), pp 840-845, 1995.