A Methodology for Applying Three Dimensional Constrained Delaunay Tetrahedralization Algorithms on MRI Medical Images

By

Feras Wasef Abutalib, B.Eng. (Computer)

Computational Analysis and Design Laboratory

Department of Electrical Engineering

McGill University,

Montreal, Quebec, Canada

December, 2007

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of Master of Engineering

© Feras Abu Talib, 2007

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-51441-2
Our file Notre référence
ISBN: 978-0-494-51441-2

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

This thesis addresses the problem of producing three-dimensional constrained Delaunay triangulated meshes from the sequential two dimensional MRI medical image slices. The approach is to generate the volumetric meshes of the scanned organs as a result of a several low-level tasks: image segmentation, connected component extraction, isosurfacing, image smoothing, mesh decimation and constrained Delaunay tetrahedralization. The proposed methodology produces a portable application that can be easily adapted and extended by researchers to tackle this problem. The application requires very minimal user intervention and can be used either independently or as a pre-processor to an adaptive mesh refinement system.

Finite element analysis of the MRI medical data depends heavily on the quality of the mesh representation of the scanned organs. This thesis presents experimental test results that illustrate how the different operations done during the process can affect the quality of the final mesh.

Résumé

La présente thèse porte sur le problème de la production de maillages tridimensionnels restreints de Delaunay à partir des images de coupe bidimensionnelles séquentielles obtenues grâce à l'imagerie par résonance magnétique (IRM). La méthode consiste à produire les maillages volumétriques des organes lus optiquement à la suite de plusieurs tâches de bas niveau, à savoir : segmentation d'image, extraction d'une partie constituante, lissage paramétrique, polissage, décimation du maillage et tétraédrisation restreinte de Delaunay. La méthode proposée permet d'obtenir une application portable que des chercheurs peuvent facilement adapter et élargir pour résoudre ce problème. L'application exige de l'utilisateur une intervention minimale et peut être utilisée soit de façon autonome, soit comme un préprocesseur d'un système adaptatif d'affinement des maillages.

L'analyse par éléments finis des données médicales de l'IRM tient fortement à la qualité de la représentation en maillage des organes lus optiquement. La présente thèse présente les résultats d'un essai expérimental qui montrent la manière dont les différentes opérations effectuées au cours du processus peuvent influer sur la qualité du maillage final.

Acknowledgements

This thesis could not have been carried out without the help from a number of persons, to whom I owe a great debt of gratitude. I wish to express my sincere respect and appreciation to Dr. D. Giannacopoulos for his support, guidance and insights in the preparation of this thesis. I must also thank my father and mother for their moral support.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Most medical imaging techniques such as X-ray, Computed tomography (CT) and Magnetic Resonance Imaging (MRI) produce high quality two-dimensional image slices. These images are very useful for the diagnostic evaluation. However, they cannot be used directly for advanced and detailed numerical analysis such as volume calculation, rapid prototyping, simulations, or treatment planning application. Even a specialized team would have difficulties to process directly the data emerging from these two-dimensional photographic image slices. Automatic tools that are able to produce geometric representations and numerical volume models of the scanned organs can provide quantitative data that aid the physicians to better treat their patients.

1.1 The Finite Element Method

The finite element method (FEM) is a tool that is used for finding approximate solutions of complex differential and integral equations [1]. The steps associated with a typical FEM can be summarized as follows: (i) Discretizing the problem region; (ii) a model of the solution is constructed over each element by an approximating function (uniquely defined by a set of parameters); and (iii) the parameters are computed based on global boundary conditions. Because of its strong foundations and favourable characteristics, FEM is sometimes considered one of the most powerful numerical analysis techniques [2].

FEM has applications in a wide range of fields in science and engineering. In the medical field, automatic Finite Element (FE) models can be extremely useful in analysing

the data of the patients. They can be utilized in many critical operations such as identifying prostate cancers, compensating for brain deformation during neurosurgery, or predicting deformation of the breast.

Mesh generation is an important initial step in any numerical FE analysis requiring high quality meshes to accurately capture the complex physical object. This step involves discretization of the three dimensional domain of the object into small elements of simple geometry such as tetrahedra or hexahedra. The mesh quality directly affects the accuracy and the efficiency of the FEM solution [25].

1.2 Motivation for the Research

Even though numerous mesh generation methods have been described to date, there are few which can deal directly with medical data input [10]. There is clearly no well established methodology for mesh generation in the medical community. Most available automatic mesh generations are targeting physical objects in general and they are not well suited to handle objects from medical images [10]. There is no doubt that a detailed investigation of the challenges and difficulties associated with automatic mesh generation of medical images needs to be explored and researched in more detail. There are many questions that should be answered such as: (i) what are the various steps needed to produce a high quality mesh from the 2D MRI slices? (ii) How does each step affect the quality of the produced mesh? (iii) How can the difficulties faced during the process be handled?

Most medical applications available in the market today are commercial and very expensive [17]. Moreover, they are designed for specific hardware architectures that limit their portability [18].

1.3 Thesis Objectives

This thesis is an attempt not only to define the current state of the technology but also to propose a complete open-source architecture that can be used to transform medical images from their native scanned format to volumetric meshes that can be used in FE analysis. The developed tool should be of great assistance to better analyze the scanned data of the patients and to aid future researchers to tackle this domain even further.

Because mesh quality is often a pre-requisite for successful finite element analysis [25], this research is supported by a detailed discussion on how mesh generation algorithms based on the constrained Delaunay triangulation principle are affected by the operations executed during the process to transform the medical images into volumetric meshes. This discussion is done with the aid of experimental test results produced by the proposed application. Also, a discussion about the possible ideas of how to improve such a process is presented.

1.4 Thesis Outline

In Chapter 2, the theoretical concepts and the art of generating the volumetric mesh from the MRI medical slices will be addressed. In Chapter 3, a proposal of an open-source system architecture that would accomplish the medical mesh generation task will be presented. In Chapter 4, the details needed to setup an environment to execute the

experiments will be outlined. In Chapter 5, the proposed architecture will be tested with real data and the results obtained will be discussed. Finally, conclusions and a summary of possible future work will be presented in Chapter 6.

Chapter 2
Theory

2.1 Overview

The two-dimensional Magnetic Resonance Imaging (MRI) slices can be used to create an image of the whole volume at once. There is no doubt that the geometric representation would contain much more information than the separated two dimensional images. High quality numerical models do not only allow the physicians to look at the region of interest from the various possible viewpoints, but can also provide them with quantitative data that can potentially increase the probability of a successful treatment. This chapter of the thesis summarizes the theoretical concepts behind the various aspects involved in producing the numerical models of the anatomical geometries used as inputs for the three dimensional finite element solvers.

2.2 Magnetic Resonance Imaging Theory

The magnetic Resonance Imagine (MRI) technique is the most common technique to produce image volumes. Images produced from this technique will be used as the inputs for our various experimentations. This section summarizes the theory behind this technology and the type of information represented in these image slices.

Unlike many other medical imaging techniques, exposure to radiation is avoided in this technique. The magnetic resonance images are not only clearer and more detailed than the other imaging methods but also can be taken from an arbitrary direction [49]. In addition, bone does not disturb the quality of the images. The produced images contain

information of a chemical nature. The different intensities in the image reflect mainly the density of hydrogen atoms and their chemical environment [3].

2.2.1 Magnetic Moment and Resonance

MRI makes use of the resonance property found in the hydrogen nucleus in the body molecules. The basic principle of magnetic resonance is that the nucleus performs spins around its axis giving it angular moment. Since the proton is a positive charge, a current loop perpendicular to the rotation axis is created, and as a result the proton generates a magnetic field $\overline{m}$ parallel to the rotation axis (Figure 2.1). The total magnetic moment is zero because the direction of these moments is randomly distributed and on average equalizes one another [3].

Figure 2.1: Visualisation of the spin and the magnetic moment [3].

When an external, uniform magnetic field ($\boldsymbol{B_0}$) is applied to the body, the hydrogen nuclei will align with the magnetic field and create a net magnetic moment, $\boldsymbol{M}$, parallel to $\boldsymbol{B_0}$. The stronger the $\boldsymbol{B_0}$ field, the greater is the total magnetisation $\boldsymbol{M}$. The applied field will also cause the magnetic moment of the nuclei to start to precess about the direction of $\boldsymbol{B_0}$ with an angular frequency $\boldsymbol{w_0}$ called the Larmor frequency - equal to [48]:

$$\boldsymbol{w_0} = \text{Gamma} * \boldsymbol{B_0} \qquad (2.1)$$

Gamma is a constant called the gyromagnetic ratio, and its value depends of the type of nucleus.

When a radio-frequency (RF) pulse (a weak rotating magnetic field), $\boldsymbol{B_{rf}}$, is applied perpendicular to $\boldsymbol{B_0}$ with a frequency equal to the Larmor frequency, $\boldsymbol{M}$ will start to tilt away from $\boldsymbol{B_0}$ as shown in Figure 2.2.

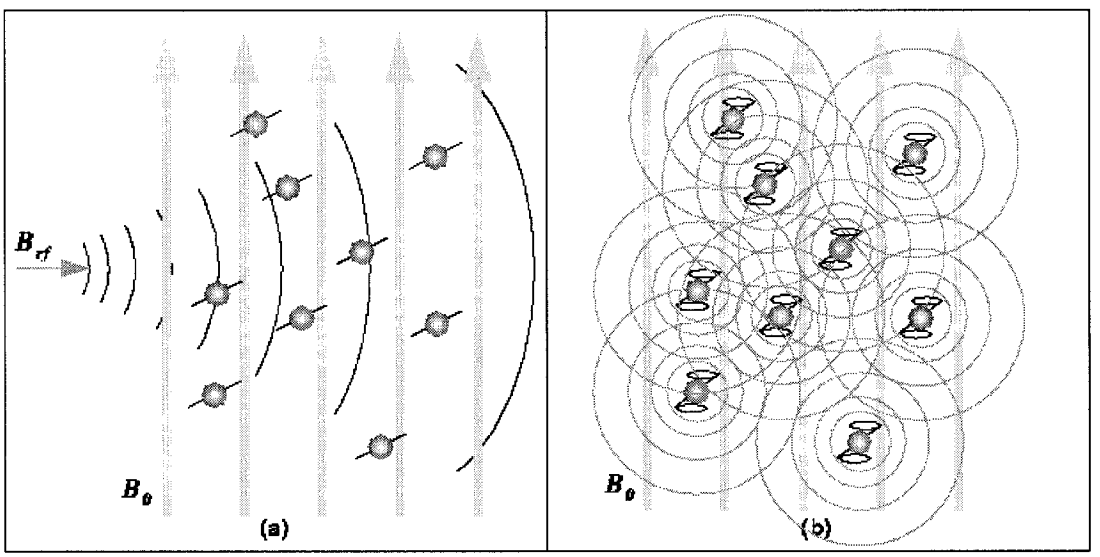

Figure 2.2: The affect of applying the RF pulse (a) and the resonance relaxation (b) [48].

When the RF signal is turned off, the nuclei return to equilibrium such that $\boldsymbol{M}$ is again parallel to $\boldsymbol{B_0}$. This return to equilibrium is referred to as relaxation [48]. During relaxation, the nuclei emit the absorbed energy as a time varying signal. This produced RF signal is measurable and measured by a conductive field coil placed around the object being imaged. This measurement is processed to obtain the grey-scale MR images.

2.2.2 Magnetic Resonance Measures

As stated, when the RF-pulse is removed, the magnetisation returns to its former state, which releases energy. At this point, the interesting information is gathered [3]:

- The energy release gives an estimate of the number of hydrogen nucleuses, which in principle is the amount of water.

- The longitudinal relaxation time T1, the time passed until the magnetisation return to "normal", gives information about the chemical surrounding of the water. The longer time, the harder the water is chemically bound.

- The transverse relaxation time T2, the time passed until the phase coherence is lost, reflects the surroundings of each individual atom, which gives a different contrast. T2 images often show differences between healthy and pathological tissue.

It can be concluded that the different intensities in the generated image reflect mainly the density of the hydrogen atoms and their chemical environment.

2.3 The Overall Model

Volume geometric models of the anatomical organs can be created from the MRI slices from a model that consists of four main steps (summarized in Figure 2.3):

Step 1: Image segmentation to generate the object boundaries of the tissue under study.

Step 2: Surface mesh generation.

Step 3: Refine the surface mesh with smoothing and decimation.

Step 4: Volume mesh generation.

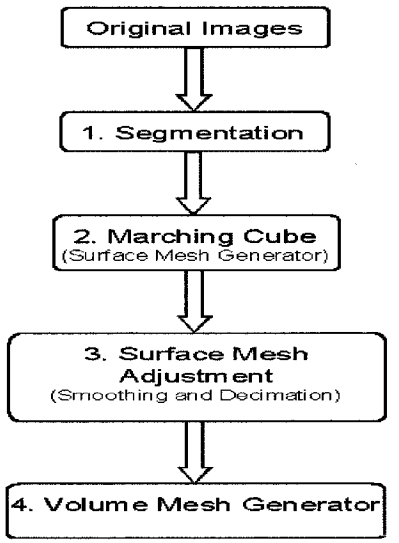

Figure 2.3: Flow chart for the creation of 3D meshes based on 2D images.

Each of these steps is discussed in more detail in the following sections:

2.4 Step One: Segmentation

As discussed in section 2.2, the produced grey-scale MRI images actually represent density of the hydrogen atoms of the different scanned organs and tissues. The first logical step in producing meshes of a specific tissue or an organ is to separate it from the other tissues/organs. Segmentation is the process that separates objects in an image.

In medical images, segmentation is the first step in creating three dimensional surfaces and volume meshes of the region of interests (ROIs) [9]. It is used to separate the different parts of the anatomical organ and to outline each ROI such that the enclosed area of the image can be identified out [28]. Segmentation by itself is used in various

medical applications. Identifying tumours is an excellent example of such application. It is an important step because it provides the initial seed set to start the meshing process.

The quality of a segmentation process can be criticized against its speed, accuracy, and degree of automation [9]. Automated segmentation of medical images is not a new area of research and it is undergoing rapid development [15]. It is a challenging task to automate the segmentation of medical images. A myriad of different methods have been proposed and implemented in recent years. In spite of the huge effort invested in this problem, there is no single approach that can generally solve the problem of segmentation for the large variety of image modalities existing today [8]. Noise in the produced image is one factor why this task is not easy to automate [9]. Another factor would be the overlapping scalar value of the different tissues which significantly decreases the effectiveness of the automatic segmenting process [9]. Pre and post processing are sometimes necessary to improve quality. Therefore, sometimes the only way to get good segmentation is to manually draw the boundaries separating the ROIs on each slice by an expert.

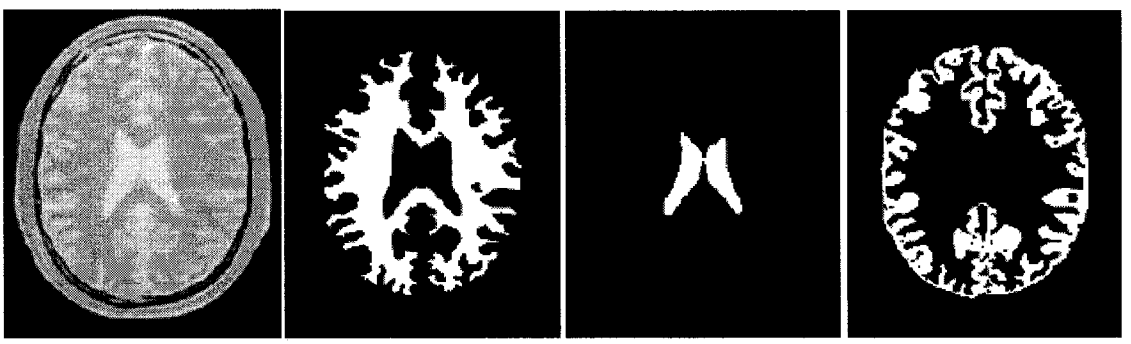

Figure 2.4: Example of the original brain MRI slice scan and its corresponding segmented images (White matter, Ventricle, Gray matter) [9].

2.5 Step Two: Surface Mesh Generation

The second step in creating three dimensional volume meshes of the region of interests (ROIs) is generating the surface mesh of the segmented data. A surface rendering algorithm is needed. Marching Cubes (MC) [4] is one of the most famous and reliable algorithms for this purpose.

The Marching Cubes algorithm is a three dimensional technique for rendering isosurfaces representation of the volumetric data. In this algorithm, an imaginary cube is used to march through pairs of adjacent segmented images by taking eight neighbour locations at a time, four vertices from each slice. A polygon is determined to represent the isosurface that passes through this cube by comparing the material type of each vertex. If a vertex has a different material from its neighbouring vertices, a boundary surface should exist between it and the others in order to separate the two different materials. There are 256 different combinations of material types that a cube's vertices could have [4]. These combinations can be obtained by reflecting and symmetrical rotation of 15 unique situations [14]. Figure 2.5 shows these possible combinations. The individual polygons are then fused into the desired surface.

The Marching Cubes combines simplicity with high speed. Despite the various advantages of using the Marching Cubes algorithm, it has some drawbacks. The surface model created by the Marching Cubes algorithm has stair-step shaped surfaces, which do not represent the natural surface curvature [8]. Also, the large density of the surface nodes and the triangles severely hinder the computational efficiency of the subsequent volume mesh generation steps [8]. This emerges the need for some surface adjustments before being able to generate the meshes.

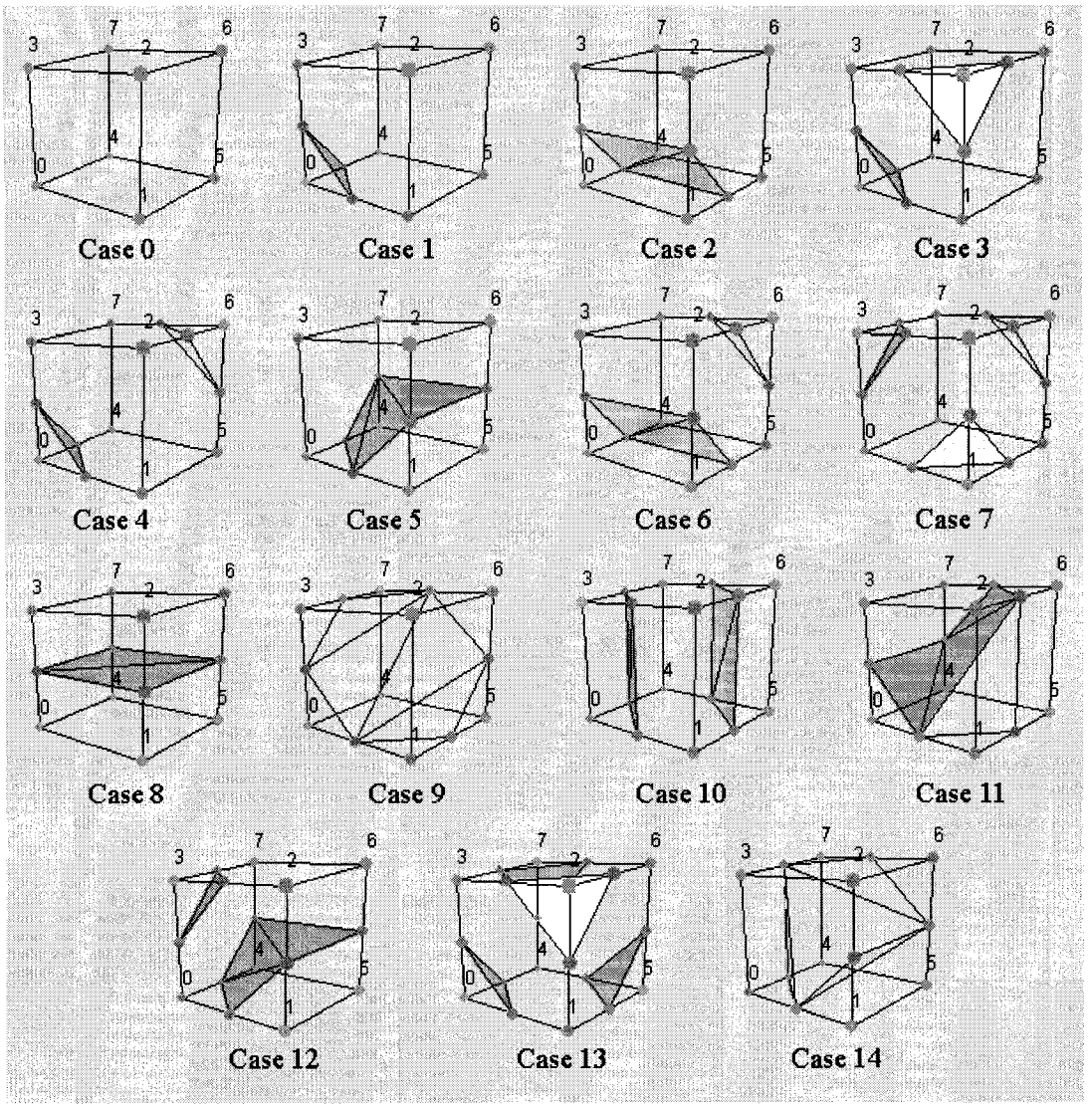

Figure 2.5: The fifteen unique MC configuration [14].

2.6 Step Three: Surface Mesh Adjustments

There are two main procedures to improve the quality and efficiency of the existing surface model: smoothing and decimation. It is worth noting that there is no preference on the order in which these procedures are applied.

2.6.1 Surface Smoothing

The main focus of smoothing is to improve the appearance of a mesh. This step is necessary to overcome the stair-step shaped surfaces produced by the MC algorithm. During smoothing, the topology of the model is not modified and only the node coordinates are adjusted. This means that the smoothing operation changes only the geometry and doesn't change the number of nodes in the existing model [16]. A common and effective technique is Laplacian smoothing [14]. The Laplacian smoothing equation for a point p_i at position $\vec{x}_i$ is given by

$$\vec{x}_{i+1} = \vec{x}_i + \lambda \sum_{j=0}^{n} (\vec{x}_j - \vec{x}_i) \qquad (2.2)$$

Where $\vec{x}_{i+1}$ is the new coordinate position, and $\vec{x}_j$ are the positions of points p_j connected to p_i, and λ is a user-specified weight. This operation can be executed on the same point repeatedly [14].

2.6.2 Surface Decimation

Decimation is a technique used to reduce the total number of the polygons in the polygonal meshes generated by the Marching Cubes algorithm [7]. The main idea is to use the minimum number of polygons without having a significant change in topology and shape of the original geometry. This will reduce the speed and the memory requirements needed to process the data emerging from the Marching Cubes algorithm [16]. The decimation technique is not domain-specific technique and it uses local operations on geometry and topology to reduce the numbers of the polygons. In this technique, three steps should be followed:

Step#1 Vertex classification: The goal of this step is to identify the vertices that are candidates for deletion. In this step, each vertex is characterized according to its local geometry and topology into one of five classifications [7, 14]:

- Simple vertex: a vertex is considered simple when every edge containing this vertex is shared by exactly two triangles and these triangles form a complete cycle.

- Complex vertex: a vertex is considered complex when one of the edges containing the vertex is not used by two triangles or if there exist a triangle that contains this vertex but not in the cycle of the triangles.

- Boundary vertex: a vertex is considered a boundary vertex when it is on the boundary of a triangle mesh and is surrounded by a semi-cycle of triangles.

- Interior vertex: a vertex is considered interior if it is a simple vertex that is used by two feature edges. An edge is classified as feature edge when the angle between the normal of the two triangles sharing this edge is greater than a specified feature angle.

- Corner Vertex: a vertex is considered corner if it is a simple vertex that is used by one, three or more feature edges.

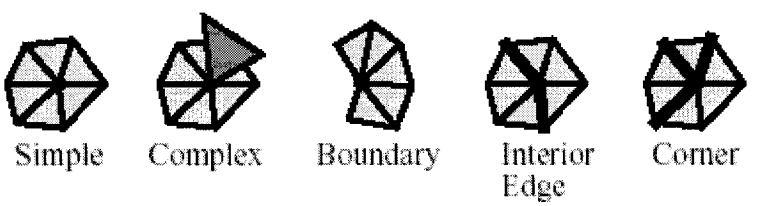

Figure 2.6: The five possible decimation vertex classifications [7].

All the vertices of type simple, boundary, interior or corner are candidates for deletion. Only the complex vertices are not considered for deletion.

Step#2 Decimation criteria: In this step, a pass over the candidate vertices should be done to determine whether the vertex along with its connected edges can be deleted and replaced with another triangulation. For the simple vertices, this is done by evaluating the distance to plane criteria. If the simple vertex is within a specified distance to the average plane then the vertex can be deleted. For the boundary and interior edge vertices, this is done by evaluating the distance to the line defined by the two vertices creating the boundaries or the feature edges. Again if the vertex is within a specified distance then the vertex can be deleted. For the corner vertices, they are usually not deleted except for the cases when the meshes contain areas with relatively small triangles that have large features angles. In these cases, the distance to plane criteria is used. When removing the triangles connected to the deleted vertex, a hole will be presented in the triangle mesh. This hole must be re-triangulated.

Step#3 Triangulation: After the vertex has been removed, the resulting hole has to be triangulated. The resulting triangles should not be intersecting with each other and not degenerating. Triangulation with good aspect ratio can better approximate the original geometry of the hole. The generic two-dimensional triangulation algorithms cannot always produce practicable results. Instead, the recursive three-dimensional divide-and-conquer technique is used to take advantage of the star-shaped of the hole. First, this hole is split into two sub-holes through the split plane. The split plane is the plane orthogonal to the average plane that contains the split line. The split line is the line connecting two non-neighbours vertices. Only if all the vertices of each of the resulted hole lie on the same side of the split plane, the split is considered acceptable. If the split is unacceptable, a different split plane should be considered. Note that if there is no possible acceptable split, then the original vertex along with its surrounding triangles will

not be removed from the mesh. This algorithm is applied recursively on the resulted sub-holes until all the resulting holes have exactly three vertices.

A successful triangulation will result in a reduction of exactly two triangles when removing a simple, corner or interior edge vertex and exactly one triangle when removing a boundary vertex [7].

2.7 Step Four: Volume Mesh Generation

A good mesh generation algorithm must not only correctly model the shape of the problem domain but also offer as much control of the sizes of the elements in the mesh [12]. Also, the generated elements should be relatively "round" because elements with large or small angles can significantly decrease the quality of the numerical solution [38].

Many possibilities are available to construct the meshes. The choice of the shape of the elements is the first thing a mesh generator should consider. Simplicial meshes is the most popular choice since its elements have simple shapes that it is easy to approximate the behaviour of a partial differential equation on each of them (0-simplex is a vertex, 1-simplex is a segment, 2-simplex is a triangle and 3-simplex is a tetrahedron etc.). This is why tetrahedra have often been used for three-dimensional analysis.

Also, meshes can be classified as structured or unstructured. In structured meshes, the indices of the adjacent nodes can be calculated with simple addition. The biggest advantage of this is that there is no storage need to store the indices of each node's neighbours. Also, this can help a lot in simplifying the parallel computation of the

problem without the use of sophisticated partitioning algorithms and parallel unstructured solvers as those needed for unstructured meshes [12].

However, structured meshes fail to properly discretize many of the problems which have irregularly shaped domains and tend to lead to a mesh with many more elements than an unstructured mesh. This leads to many more redundant computations that do not result in a more accurate solution. The unstructured mesh fits itself more nicely into these domains. Experimentation has proven that unstructured meshes are much better than structured meshes and can provide multi-scale resolution and conformity to complex geometry [38].

2.8 Delaunay Triangulation Mesh Generation

2.8.1 Overview

The most popular triangulation meshing methods are those that utilizing the Delaunay "empty sphere" criterion. This property says that any node in the mesh must not be contained within the circle/circumsphere of any triangle/tetrahedron within the mesh [32]. Figure 2.7 is a simple 2D example of this criterion. Delaunay mesh generation algorithms are both provably good and very practical [38].

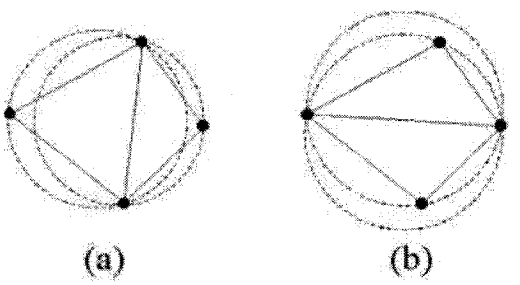

Figure 2.7: Illustration of the Delaunay criterion. (a) maintains the criterion while (b) does not [50].

Among all triangulations of a vertex set, the Delaunay triangulation maximizes the minimum angle in the triangulation, minimizes the largest circumcircle, and minimizes the largest min-containment circle, where the min-containment circle of a triangle is the smallest circle that contains it (and is not necessarily its circumcircle) [12]. This nice property is the reason for the wide use of Delaunay triangulation in mesh generation. It is important to note that the max-min property holds only for the two dimensional Delaunay triangulation and doesn't apply to three dimensions and higher triangulations. In spite of that, experimental results show that the Delaunay triangulation in three dimensions and higher is still invaluable for mesh generation applications [12].

A standard way to construct Delaunay meshes is to first obtain an initial set of nodes by meshing the boundary of the given geometry. These nodes are then used to get the initial Delaunay triangulation. More nodes are then added incrementally to this triangulation structure. The resulted tetrahedrons are redefined locally to maintain the Delaunay criterion. There exist many methods to determine where to locate these interior nodes.

The simplest approach for point insertion is to define the nodes from a regular grid of points covering the domain at a specified nodal density [50]. A sizing function, which is determined by the user, can be defined in order to provide changeable element sizes and the nodes are inserted until the condition of sizing function is met. Another approach can be done by recursively inserting the new nodes at the tetrahedrons' centroids [29]. The third approach is to follow a specific order in inserting the new nodes at element circumcircle/sphere centers. This technique is called "Guaranteed Quality" as triangles can be generated with a minimum bound on any angle in the mesh [31]. The line segment between the circumcircle centers of two adjacent triangles or tetrahedrons

is referred as "Voronoi segment". A new node is established at a point along the segment to satisfy the best local size criteria. This method produces very structured meshes with six triangles at every internal node [32, 50].

2.8.2 Boundary Constrained Triangulation

Even though the minimum angle of the Delaunay triangulation in two dimensions is maximized, strictly Delaunay triangulation is still not enough to produce good mesh elements. These methods are unaware of the requirement to maintain an existing surface triangulation. As a result, these boundary requirements may not appear in the final triangulation. Secondly, these methods may produce quite poor triangles/tetrahedrons. Certain simplices, according to their geometry, can result in a significant decrease in the precision of the solution. In two dimensions, for example, a triangle which is too "flat" (a flat triangle has a large angle) leads to large errors (See Figure 2.8). To prevent this from happening, the mesh generation has to produce elements which are as close of being regular tetrahedrons as possible.

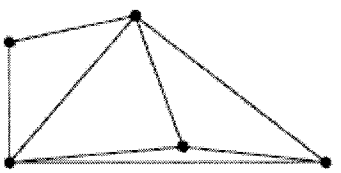

Figure 2.8: The Delaunay triangulation of a set of vertices. It does not usually solve the mesh generation problem, because it may contain poor quality triangles (bottommost triangle) [12].

Both of these problems can usually be solved by inserting additional vertices into the triangulation. Therefore, many implementations usually add a second step to recover the

required surface triangulation. By doing that, the triangulation may no longer be strictly Delaunay, hence the term "Constrained Delaunay Triangulation" [12]. A constrained Delaunay simplex is one whose circumscribing circle can contain other vertices as long as they are hidden by other input simplices of order one less than the dimension we're working in [38].

A Constrained Delaunay Triangulation (CDT) is one which allows constrained Delaunay simplices to be included in it and therefore has a larger degree of freedom. Although not strictly Delaunay, a CDT has similar properties as the Delaunay triangulation. For instance, given a set of segment inputs, a CDT will have the maximum minimum angle out of all the other triangulations that conform to the input segments and vertices [12]. Certain triangles, however, may still be deemed "bad" because they may have small angles, as imposed by input segments which are at small angles of each other [12].

In two dimensions, the edge recovery is relatively straightforward. The edges of a triangulation may be recovered by iteratively swapping triangle edges. The process is considerably more complex in three dimensions. Two different methods are presented here for recovery of the boundary.

In the first approach, a series of tetrahedral transformations is done by swapping two adjacent tetrahedra for three in order to recover the edges as shown in Figure 2.9. It is to be noted that if a swap cannot resolve the edge, more nodes must be added. After recovering the edges, additional transformations are executed in order to recover the face. That is done by swapping three adjacent tetrahedra at an edge for two. In case that the surface facet can not be resolved through the transformation, more complex transformations or additional nodes can be added during the face recovery phase [37].

Two phases are included in the second approach: an edge recovery phase and a face recovery phase [29]. In this approach, nodes are inserted directly into the triangulation wherever the surface edge or facet cuts non-conforming tetrahedra. This process temporarily adds additional nodes to the surface in order to facilitate the boundary recovery. However, these nodes will be deleted once the surface facets have been recovered, and the resulting local void re-triangulated.

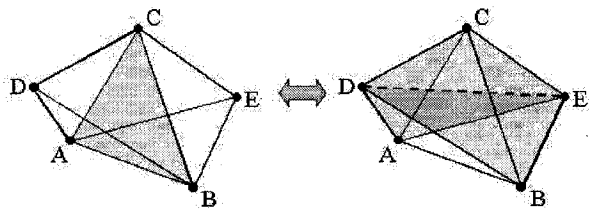

Figure 2.9: Tetrahedral transformation where two tetrahedrons are swapped for three [50].

2.8.3 Delaunay Mesh Refinement

The resolution of the unstructured meshes can vary throughout the mesh. For example, around a singularity, tetrahedra should be small and should become larger and larger as we get farther from the singularity. In order to obtain size-variable meshes, refinement algorithms are used. Once the solution is calculated using an initial coarse mesh, areas with large error can be identified and then further partitioned into smaller simplices in order to gain higher precision. Refinement can be defined to be any operation performed on the generated elements in the mesh that effectively reduces the local element size [38]. In fact, post-processing refinement is almost always needed to improve the overall quality of the elements generated. Element refinement procedures are numerous.

Smoothing is one of the most popular techniques for refinement. Smoothing adjusts node locations while maintaining the element connectivity. With smoothing, there are no changes made to the topology of the mesh. Another method for refining is called "clean-up". In this method the process changes the element connectivity within the mesh. In refining the mesh, it is important to keep the Delaunay property of the mesh so that no "bad" triangles are formed in the process. Incremental algorithms for mesh refinement should add vertices one by one while keeping the mesh Delaunay [38].

Chapter 3
The System Architecture Proposal

3.1 Objective

One of the objectives of this thesis is to propose an architecture that can be used to transfer the medical images into meshes that can be consumed by the FEM software such as ANSYS, ABAQUS, and NASTRAN.

The proposed system will also be initiated and utilized to supply us with experimental data results that can be used to enrich the discussion of the effect of some of the different steps on the quality of the generated meshes.

3.2 Requirements:

3.2.1 Functional Requirements:

The following list is a summary of the main functional features that the proposed system should have:

I. The system should provide the ability to process MRI images in their native format.

II. The system should be capable of providing the various image processing services needed during the various steps to generate the mesh. These services should include the segmentation, isosurface creation, smoothing, and decimation techniques.

III. The system should provide the ability to generate volume adaptive Delaunay meshes when seeded with an initial surface mesh.

IV. The system should provide a visualization service. This should include elementary functions as zoom, rotate, and translate for the 2D and 3D views of the processed data.

Figure 3.1 shows the building blocks of the proposed architecture functionalities based on the previously listed functional requirements.

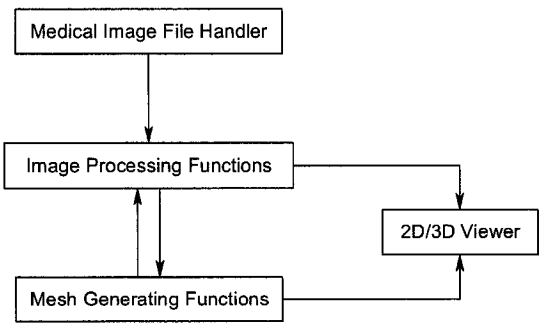

Figure 3.1: The proposed system architecture functionalities.

3.2.2 Non-Functional Requirements:

The following list is a summary of the main non-functional features:

I. Performance: The proposed system should be designed to maximize speed while minimizing the memory as much as possible. This requirement is important because the computational cost associated with processing medical data tends to be very expensive.

II. Usability: The proposed system should be easy to learn so that new users can start using the system with minimal knowledge.

III. Expandability: the proposed system should be expandable so that new functionalities, features, and components can be easily developed and integrated into the system.

IV. Portability: The system should be portable across the different platforms and should not be limited to a specific hardware. Also, it shouldn't rely on proprietary libraries that would prevent public researchers from taking advantage of it.

3.3 Input Representation

As the first functional requirement stated, the system should provide the ability to process MRI images in their native format. Digital Imaging and Communications in Medicine (DICOM) files is the most popular standard for sending/receiving medical scan images. It was created by the National Electrical Manufacturers Association (NEMA) [51]. DICOM is a comprehensive set of standards for handling, storing and transmitting information. Unlike The previous attempts at developing a standard, DICOM had the potential to actually achieve its objective for transferring images as well as associated information between devices manufactured from various vendors.

A DICOM file contains both a header and the image data. The header is used to store metadata information such as the patient's name, the type of scan, image dimensions. The DICOM image data can be stored as raw data or compressed using lossy or lossless variants of the JPEG format to reduce the image size.

The proposed software system should be able to handle the different DICOM medical files variations as these image files supply the input data to our system.

3.4 Software Components and Libraries

Writing software from scratch that would achieve the listed functional requirements is a very difficult task that requires extensive amount of effort and time. An alternate approach would be to look for already developed open-source software components/libraries of which each would be able to achieve a sub-set of the functional requirements. These software components are then connected together to achieve all the proposed functional requirements. There are many components available. The idea is to try to use the minimum number of components while achieving the maximum possible number of functionalities. It is important to note that during the search for such components, one has to always not overlook the non-functional requirements. For example, a component might exist that would accomplish the required functionality but would break the portability non-functional requirement. Furthermore, the chosen components should be compatible with each other to some degree. This would make the integration process between them feasible and doesn't require a huge effort.

In this section, the chosen components/libraries needed to accomplish the system proposal are introduced. A description of each component is presented along with a discussion on why this component was chosen and how it should help in achieving the overall proposed requirements. Figure 3.2 shows the overall proposed object design process and how the different chosen components communicate with each other in order to accomplish the various tasks and functionalities.

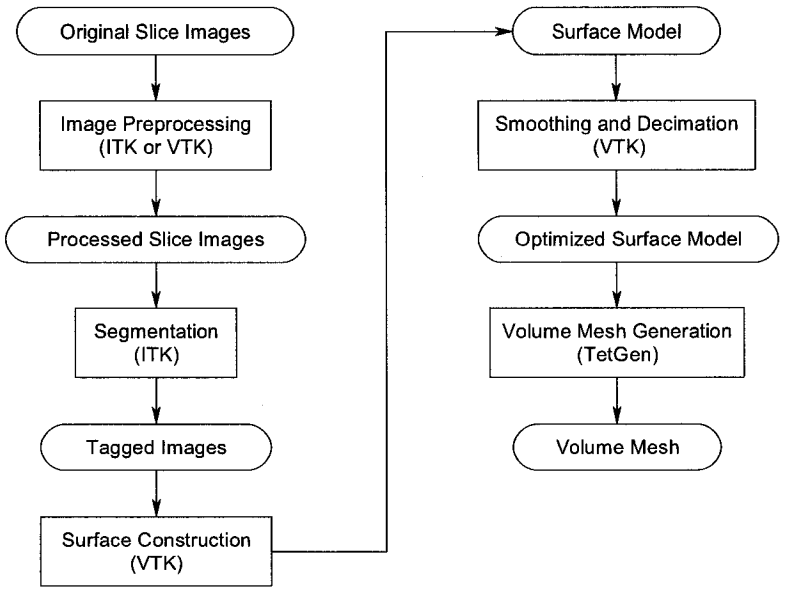

Figure 3.2: The proposed object design process with the data flow.

3.4.1 The Insight Toolkit

The Insight Toolkit (ITK) is an open-source and object-oriented software system library for performing registration and segmentation on images. ITK also provides functionalities that allow reading DICOM files. Therefore, this software component can be used to help fulfill the functional requirements I and II.

ITK is implemented in C++. It is cross-platform and uses a build environment known as CMake to manage the compilation process in a platform-independent way (see section 3.6). ITK was started in 1999 under a contract by the US National Library of Medicine. This Project is still under heavy development to include even newer segmentation algorithms and techniques.

ITK is very large and relatively complex compared to the other components. ITK's C++ implementation style is referred to as generic programming. It uses templates for both

the implementation of the algorithms and the class interfaces themselves. This type of heavily templated C++ code challenges many compilers and it can take much longer to compile.

On the other hand, the templates programming allows the same code to be applied generically to any class or type that happens to support the operations used. Such C++ templating means that the code is highly efficient, and that many software problems are discovered at compile-time, rather than at run-time during program execution.

3.4.2 The Visualization Toolkit

The Visualization Toolkit (VTK) is an open source software library for 2D/3D image/surface processing, and visualization. This software component can be used to help fulfill the functional requirement II and IV. VTK supports a wide variety of visualization algorithms and advanced modeling techniques such as surface construction and polygon reduction (Decimation). The VTK library is portable because it has been installed and tested on nearly every Unix-based platform, PCs (Windows 98/ME/NT/2000/XP), and Mac OSX Jaguar or later [43].

One should understand the pipeline architecture used in VTK in order to take full advantage of it. In this architecture, multiple elements are attached together to perform a complex task. Typical pipeline architecture is outlined in Figure 3.3.

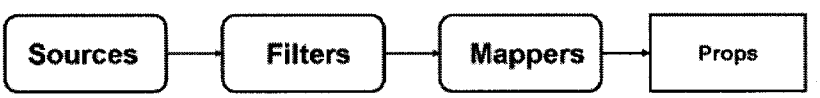

Figure 3.3: A typical pipeline architecture in VTK.

In the figure, the sources are the classes that produce the data while the filters are the classes that operate on the data to produce a modified version of it. The mappers, on the other hand, are the interface classes between the data and the graphics primitives. Multiple mappers may share the same input, but render it in different ways. The props are the classes needed to generate the visible representation of the output of the mappers on the screen [13].

The design and implementation of the VTK library has been strongly influenced by the object-oriented principles and therefore understanding of the object oriented principles will help to utilize the VTK underlying classes more effectively.

3.4.3 TetGen Mesh Generator

The functional requirement III can be accomplished by the TetGen software component. The main goal of TetGen is to generate suitable Delaunay tetrahedralization, constrained Delaunay tetrahedralization and quality tetrahedral meshes for solving partial differential equations by finite element or finite volume methods. TetGen code is highly portable since it is written in ANSI C++ and therefore can be compiled in Unix/Linux, Windows, and MacOS [45].

The final step of generating the volume meshes from the adjusted surface meshes is the most time and memory consuming in the overall process. It was reported that the TetGen implementation is fast and memory efficient. For example, on an Apple laptop (2.16GHz Intel Core 2 Duo); it takes 2.38 seconds to compute the Delaunay tetrahedralization of 40,000 randomly distributed points with 9.4MB heap memory. For one million points, it uses 93 seconds and 234.47MB heap memory [45]. The TetGen

software component was chosen for one more reason and this is because it can perform efficient mesh refinement by inserting new vertices to improve the overall mesh quality. This can be utilized to refine the generated mesh at places where the error is too large.

3.4.4 The Scripting language Tcl

3.4.4.1 Tcl Rational

Tcl is an open-source interpreted language. It is available on a wide variety of platforms, including Windows, Mac, and essentially all flavours of UNIX (Linux, Solaris, IRIX, AIX, *BSD*, etc.) [47]. There exist many interpreter shells such as tclsh and wish which can execute the various Tcl commands. This scripting language will be used primarily to achieve the binding glue role in the proposed system. The reason why a scripting language was chosen to accomplish this role instead of a traditional compiled language, like C++, is mainly because it is easier to learn a scripting language and it is more readable. Remember that if a language is chosen to be the binding glue between different components, then this language will be used to control the order in which the operations of the other components are executed. It will be the interface from which new experiments can be written and old experiments can be reorganized. Because of the use of a scripting language, there will be no need to recompile the code to execute the new/modified experiments.

On the other hand, one would argue that a scripting language would hinder the performance of the system. This would be true if the scripting language is being used to carry out the costly computational operations. Remember that these expensive operations will be executed by their corresponding components and not by the scripting language itself. The scripting language role will be just to provide an easy-to-use interface to call these expensive operations.

	A traditional compiled language	A scripting language
Examples	C++, Java	Tcl, Perl, Python
Application	Large and complex applications where performance is important	Simple and small to medium applications where performance is less important
Structure	Very structured	Minimum structure, less overhead, easy to interchange

Table 3.1: Comparison between a scripting language and a traditional compiled language.

But why was the Tcl language chosen and not any other scripting language such as Python, Ruby, or Perl? From the extendibility point of view, the Tcl language is the one with the best integration with the graphical user interface toolkit "Tk" [13]. This would allow integrating a UI component easily in the system if needed. In fact, most of the other languages, directly or indirectly invoke Tk and hence an understanding of Tcl is helpful in learning these languages as well [13].

3. 5 Software Interfaces:

The overall functionalities can be achieved by combining together the different software libraries and components. However, the Interfaces between the different components should be defined first to facilitate this process. This section establishes these interfaces which would make the integration process possible.

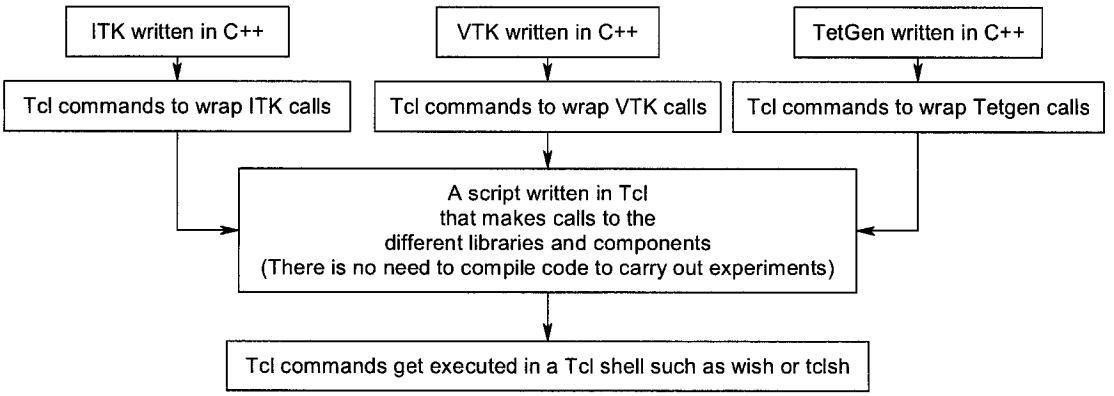

Figure 3.4: The proposed interfaces between the software components.

3.5.1 ITK-VTK interface

Fortunately, both ITK and VTK have an automated wrapping process to generate interfaces between the implemented C++ class library and the interpreted programming languages Tcl. However; this is not enabled by default and customized changes to the settings to compile their source code should be made to enable this feature (Refer to Appendix I).

Also, ITK and VTK use different wrapping system and therefore it is not a straightforward procedure to convert the output of the ITK engine to be used directly as input by the VTK engine from within the Tcl command. There are mainly two possible approaches to solve this problem: (i) convert the output of ITK engine to a common file format that can be read by the VTK engine; or (ii) make use of the image data importer and exporter classes that have been implemented in both engines. The first approach is simpler as there is no challenge in instantiate image importer classes in the VTK without knowing first the pixel type of the produced image of the different ITK segmentation techniques. On the other hand, the first approach has worse performance as IO operations are involved. Again this should not be an issue as these types of operations are not the

bottleneck operations in such medical application. The common file format that can be produced or consumed by both engines is the .vtk file format.

3.5.2 VTK-TetGen Interface

There is no automated wrapping process to generate an interface between the implemented C++ TetGen class library and the interpreted programming language Tcl. This wrapper would make the ultimate use of the scripting and compiled language combination in a single piece of software (hybrid programming approach) [13]. It would allow the TetGen code to be called directly from within a Tcl shell and therefore maintaining the Tcl interface among all the components of the proposed system. In fact, writing such code is a necessity to complete the integration process and to provide a mechanism for future code changes to be integrated into the proposed system regardless of the language they are written in. See Appendices III and IV for the C++ structure for such wrapper.

Also, a similar approach to the ITK-VTK interface is being used to transfer the data from the VTK engine to the TetGen engine by using the .ply common file format that can be produced by the VTK engine and consumed by the TetGen engine. It is important to note that TetGen can only accept .ply files as input data, and it cannot produce them as output data. Therefore, code was written to transfer the TetGen output files into a format that can be consumed by the VTK for visualization and mesh quality reporting purposes (Refer to Appendix V).

3.6 Software Portability:

Writing software that can compile and run on different operating systems is not an easy task even for the most experienced programmer [13]. This is especially true for software written with the C++ language. C++ programmers not only have to make sure to follow the standard C++ coding rules and to restrict their use to the standard libraries but have also to provide a mechanism for their code to be compiled and built across the different platforms.

CMake is a cross-platform and open-source make system. This tool can help in controlling the software compilation process of a software component across the different platforms. Portability of C++ code is easier with the use of such tool. CMake is quite sophisticated. It is possible to support complex environments requiring system configuration, pre-processor initializations, code generation, and template instantiation. CMake takes as an input a set of configuration files and generates as an output native makefiles (Unix) or Visual studio projects (MS-WINDOWS) for the application. Because of this tool, there is no longer a need for the programmer to write both by hand.

Both VTK and ITK provide the compiler independent CMake configuration file consumed by the CMake executable to generate the VTK/ITK code according to the chosen platform/compiler environment. CMake configuration file was produced for the TetGen code to be able to achieve the same functionality (See Appendix II).

Chapter 4
Experiments Setup

This chapter of the thesis builds the foundations needed for the experiments to be carried out. This includes how to evaluate the results of the experiments, how to setup the hardware and software environments, and specifying the input data to the experiments.

4.1 Mesh Quality Evaluation

An important step in conducting numerical experiments is to setup the criteria with which one can evaluate the results of the experiments. This can be summarized into three categories: (i) Geometric mesh quality measures; (ii) Mesh surface approximation measures; and (iii) Performance measures.

No matter the context, it is clear that mesh quality evaluation are essential for achieving optimal accuracy and efficiency of the finite element analysis [21].

4.1.1 Geometric Mesh Quality Measures

It is well known that the accuracy and efficiency of the finite element method can be directly affected by very poor quality elements [2, 21-23]. A single poor quality element can cause the slowdown of iterative solvers and large round-off error in the finite element solution [21]. The tetrahedron simplex is the most flexible element for covering complex topologies in three dimensions because the complete polynomial expansion functions can be defined over tetrahedra with relative ease [26, 27]. Tetrahedra are considered a good choice to represent medical data since it is particularly

straightforward to use them to describe smooth surfaces [6]. Geometric mesh quality measures evaluate the shape of tetrahedra based on purely geometric characteristics. Most of these measures are usually presented in the form of quantities such as volume, edge lengths and radii of the spheres associated with the tetrahedral elements. The most common used benchmark test for geometric quality measures are based on a specific distortion of an equilateral tetrahedron [33, 34]. Tetrahedra that are particularly skewed slow down and produce errors in the solutions given by partial differential equations solvers. The reason for these problems is that the equations corresponding to skewed tetrahedra can be very poorly conditioned [6].

According to the dihedral angle, the angles between triangles in a tetrahedron, poorly-shaped tetrahedra can be classified into five classes [6] as shown in Figure 4.1. A needle tetrahedral has the edges of one triangle much smaller than the other edges. A wedge has one edge much smaller than the rest. A sliver has four well separated points that nearly lie in a plane. A cap has one vertex very close to the triangle spanned by the other three vertices. A spindle has one long edge and one small edge.

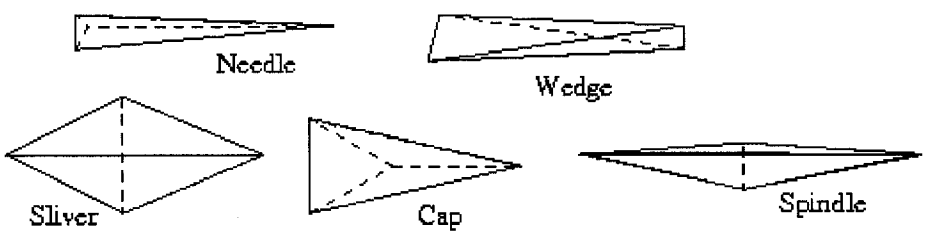

Figure 4.1: The five classes of poorly-shaped tetrahedra [6].

Both α and β_{Baker} measures are considered among the most commonly used measures in identifying tetrahedra distorted from the equilateral shape.

$$\alpha = \frac{3R_i}{R_c} \tag{4.1}$$

R_i is the radius of the insphere inscribed in a tetrahedron such that each face of the tetrahedron is tangent to the sphere and R_c is the circumsphere radius of the sphere passing through all four vertices of the tetrahedron. The α ratio achieves a value of one for equilateral tetrahedra [33]. Elements having a lower value considered poorer quality.

$$\beta_{Baker} = 2\sqrt{6}\frac{R_i}{l_{\max}} \tag{4.2}$$

$l_{\max}$ is the length of the largest edge in a tetrahedron. Similarly, this ratio achieves a value of one for equilateral tetrahedra and a lower value for the tetrahedra distorted from the equilateral shape. Note that the normalized Baker quality criterion has a solid theoretical basis linking it to approximation accuracy for first-order tetrahedral finite element [35]. In particular, the measure is fair and associated with interpolation error bounds suggested by approximation theory [25].

Therefore, the quality of the shape of the mesh elements generated by the experiments can be compared by computing α and β_{Baker} measures. Also, reporting the minimum, maximum and average minimal dihedral angle of the generated tetrahedrons would also help to visualize the quality of the mesh elements [12].

4.1.2 Mesh Surface Approximation Measures

The non-equilateral tetrahedra are known to negatively impact finite element accuracy and efficiency; however, it is not sufficient for evaluating tetrahedral meshes in the finite element context [25]. One of the most important properties of a tetrahedral mesh is that it must completely fill the region being simulated. In the proposed methodology, this heavily depends on the surface approximation of the anatomical object. The quality of the surface approximation can be evaluated using the Hausdorff distance mean error $d_m(S,S')$ [19] between the approximated surface (S') and the original surface (S).

$$d_m(S,S') = \frac{1}{|S|}\iint_{p\in S} d(p,S')dS \tag{4.3}$$

|S| donates the area of the surface S and $d(p,S')$ is the distance between a surface S' and a point p belonging to surface S and it can be computed as

$$d(p,S') = \min_{p'\in S'} \|p - p'\|_2 \tag{4.4}$$

Note that $\|.\|_2$ donates the usual Euclidean norm. As can be seen in Figure 4.2 the distance between a point p and a surface S' is defined to be the distance between the point p and the nearest point in S' to p.

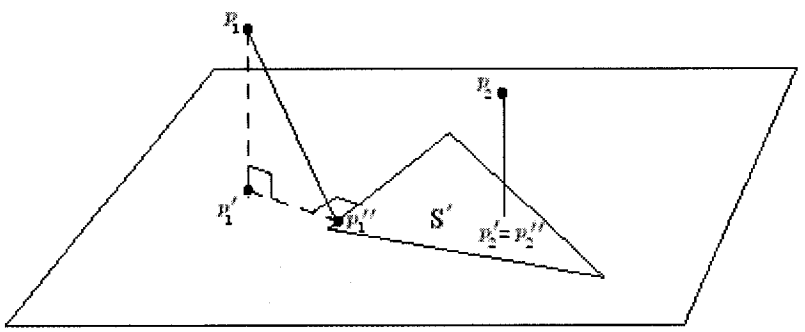

Figure 4.2: Illustration of how to compute the distance between a point and a surface.

A natural variant of the equation 4.3 would be the root-mean-square error version which can also be used as a measure for the accuracy of the surface approximation [19].

$$d_{rmse}(S,S') = \sqrt{\frac{1}{|S|}\iint_{p\in S} d(p,S')^2\, dS} \tag{4.5}$$

It is worth noting that the definition of the distance between two surfaces is given by the Hausdorff equation 4.6.

$$d(S,S') = \max_{p\in S} d(p,S') \tag{4.6}$$

This equation is not symmetrical (i.e. $d(S,S') \neq d(S',S)$, see Figure 4.3) and therefore equation 4.3 is not symmetrical either. Symmetrical version of equation 4.3 can be derived as follows:

$$d_{ms}(S,S') = \max\left[d_m(S,S'), d_m(S',S)\right] \tag{4.7}$$

Similarly, a symmetrical version of equation 4.5 can also be derived. It is important to mention that the symmetrical version provides a more accurate measurement of the error of the surface approximation since the value of each distance side can be largely different than the other as illustrated in Figure 4.3 where $d(A,S') << d(B,S)$.

Note that the original surface can be the surface produced by the marching cubes algorithm, before decimation. In this case, the approximate surface would then be the surface produced by decimation.

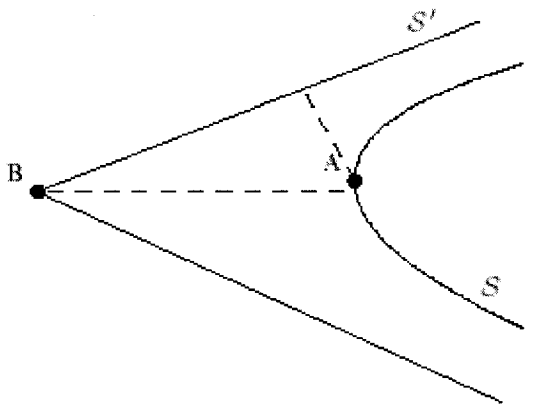

Figure 4.3: Symmetrical distance between two surfaces [19].

4.1.3 Mesh Performance Measures

The challenge of anatomical modeling from the medical images is not only to accurately model complex shapes but also to reduce the modeling time [20]. The produced mesh that leads to as few computations as possible in a finite element simulation is highly desirable as intra-operative medical procedures require the mesh generation process to be very fast. While some methods can perform well with large datasets, it is often reported that the execution time can be very lengthy [20]. Both the bad quality and the large number of the mesh elements can negatively affect the execution time of a FEM simulation [5]. While the shape quality should be high in order to produce accurate results, the number of the mesh elements should be small enough to keep the computation time reasonable [24]. Consider for example, the typical Marching Cubes algorithm would generate a very large number of polygons: one to three million triangles from a 512x512x512 volume. If these generated polygons are used directly to generate the mesh then a large number of mesh elements would be produced without necessarily increasing the quality of the mesh. However, one should remember that the number of elements cannot be very small as this would lead to mesh elements with large sizes.

Mesh elements with large sizes are reported to produce less FEM accurate results than the smaller ones [12].

Another performance attribute that should be considered is the memory needed during the process. Mesh generation algorithms might require huge amount of memory especially when the number of mesh elements generated is large.

Therefore, it can be seen in order to evaluate the performance of the mesh generator: one would report the time needed for the various tasks to be executed, the number of mesh elements that were generated, the sizes of these elements, and the memory needed during the process.

4.2 Environment of the Experiments

4.2.1 Hardware Environment

The proposed system is a highly portable solution that is not limited to a specific hardware system. The experiments in this thesis were performed on an Intel Pentium 4 workstation equipped with a 2.0 GHz CPU, 512KB cache size, and 512MB physical memory.

4.2.2 Software Environment

Many different software components and tools were used in these experiments. Table 4.1 summarizes the version numbers of the different software components.

Software component	Version#
OS	Windows XP Sp2
Tcl	8.3
Visualization ToolKit (VTK)	5.0.3
Insight Toolkit (ITK)	3.4.0
TetGen Mesh Generator	1.4.2
CMake	2.4 patch 7
C++ Compiler	Microsoft Visual C++ 6.0 SP6

Table 4.1: Software environment of the experiments.

Refer to Appendix I for the full guide of the detailed technical instructions on how to setup these different components together.

4.3 Input Data

The experiments were carried out on a dataset derived from a frog. The data was originally obtained from the Virtual Frog project of the Lawrence Berkeley National Laboratories [42]. The intention of the experiments done on the frog data was to generate a better understanding of how to transform the image slices into volumetric mesh elements using the proposed system architecture by going through the different low-levels tasks.

Chapter 5
Experimental Results

There are many steps involved in the process to generate volumetric medical meshes. In this chapter, experiments were carried out in order to study the effect of some of these steps on the quality of the final produced meshes.

5.1 The Reference Case

The steps depicted in Figure 5.1 will be carried out on the segmented image slices of a frog. The names of the functions that are actually called are also presented in the figure. Note that some customized code had to be written to make the flow in the figure possible. Refer to Appendices III-VIII for the more technical details.

The idea is to evaluate the effect of the operations involved on the quality of the final mesh generated. The first step would be to execute the suggested set of operations depicted in Figure 5.1 with some default parameter values on a set of different tissues of a frog and report back the quality of the meshes that are generated. After that, the experiments will be repeated by modifying the parameter values of the operations, replacing the operations with equivalent ones that implements different algorithms, or by adding new operations in the process. A comparison of the results will be done and discussed throughout the different experiments.

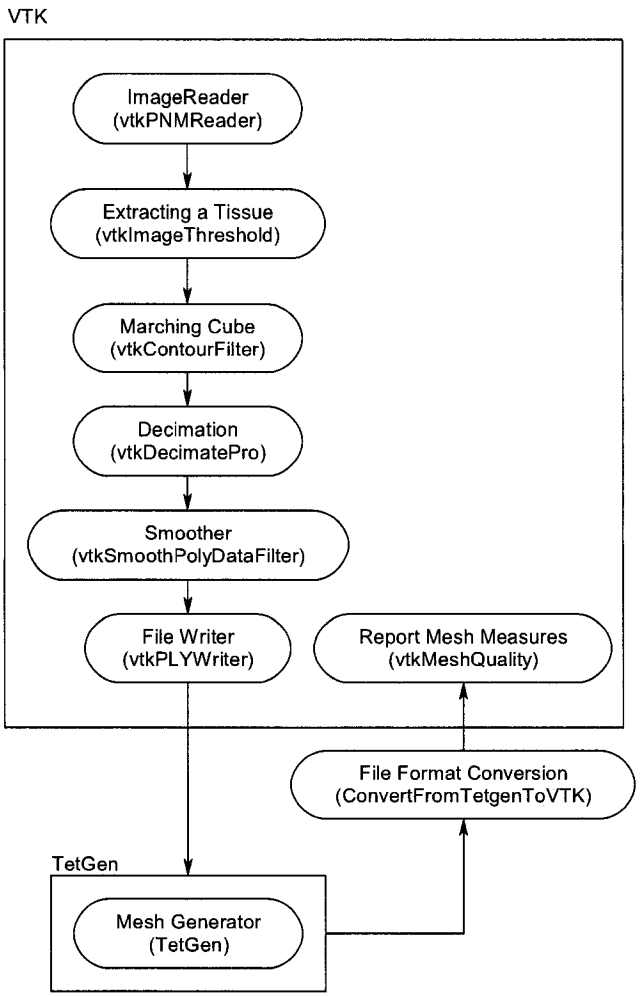

Figure 5.1: Flow chart of the various procedures of the experiment.

Refer to Appendix VIII for the experiment.tcl script file which describes how the operations are actually called. Note that each operation has a section in the file which allows the user to enable/disable the operation and/or change the parameter values of the operation. The results on applying the operations of Figure 5.1 on eight different tissues of the frog are presented in Tables 5.1 and 5.2.

Table 5.1: Numerical results for the geometric mesh quality measures of the reference case.

Tissue Name	Dihedral Angle			α			β_{Baker}		
	Min	average	Max	Max	average	Min	Max	Average	Min
brain	0.823	43.571	88.761	0.999	0.538	0.013	0.977	0.479	0.020
eye_retina	0.389	43.442	89.130	0.999	0.517	0.000	0.986	0.476	0.003
eye_white	0.125	42.586	89.044	0.999	0.478	0.001	0.985	0.446	0.005
heart	1.055	43.545	89.018	0.998	0.550	0.005	0.979	0.490	0.014
kidney	0.152	43.378	88.979	1.000	0.293	0.000	0.988	0.475	0.002
l_intestine	0.872	43.347	89.448	0.998	0.529	0.002	0.977	0.480	0.011
lung	0.116	43.292	89.129	0.999	0.299	0.000	0.980	0.460	0.002
spleen	0.888	42.975	88.866	0.995	0.523	0.015	0.954	0.470	0.016

Table 5.2: Numerical results for the performance mesh quality measures of the reference case.

Tissue Name	# of Tetrahedron generated	Tetrahedron Volume			Mesh Generation Time (s)	Memory (KB)
		Min	average	Max		
brain	52515	0.00011	0.544	32.315	6.124	5796
eye_retina	133186	0.00000	0.416	9.812	22.856	14972
eye_white	51318	0.00002	0.465	25.172	6.092	5925
heart	84468	0.00011	0.655	33.424	11.449	9341
kidney	113752	0.00000	0.514	19.914	18.437	12824
l_intestine	93577	0.00004	0.820	78.886	11.868	10360
lung	119058	0.00000	0.594	39.445	17.261	13371
spleen	13012	0.00018	0.443	13.132	1.325	1505

Note that the results in both tables were obtained when the target reduction rate parameter for the decimation operation was set to 0.6. The values of the smoothing factor and the number of smoothing iterations were set to .01 and 10 respectively. The TetGen engine was instructed to generate a quality constrained Delaunay tetrahedralization for the surface mesh produced with a constraint to have a minimum radius-edge ratio of value 1.414 (switch pq1.414). This set of parameters is going to be considered the experiment's reference case to which the parameters of the other experiments can be compared.

It is worth noting that the proposed application reports a significant amount of useful information regarding the various operations during/after their executions. This is done by leveraging the reporting functionalities offered by the various components. This includes for example the use of the vtkMeshQuality object in the system and customizing it to compute the required mesh measures. The verbose logging feature of the TetGen component is another opportunity to enhance the system's reporting capabilities. Refer to Appendix VII for a sample output when the operations in Figure 5.1 get executed.

Also, Figure 5.2 shows a visualization example on how the data are transformed during the various steps to create the final volume mesh of one of the frog's tissues, the spleen. Note that the adjusted surface mesh produced after the decimation and the smoothing operations fixes the stair-step shaped surface produced by the Marching Cubes operation and reduces the number of the triangles needed to represent the surface as well.

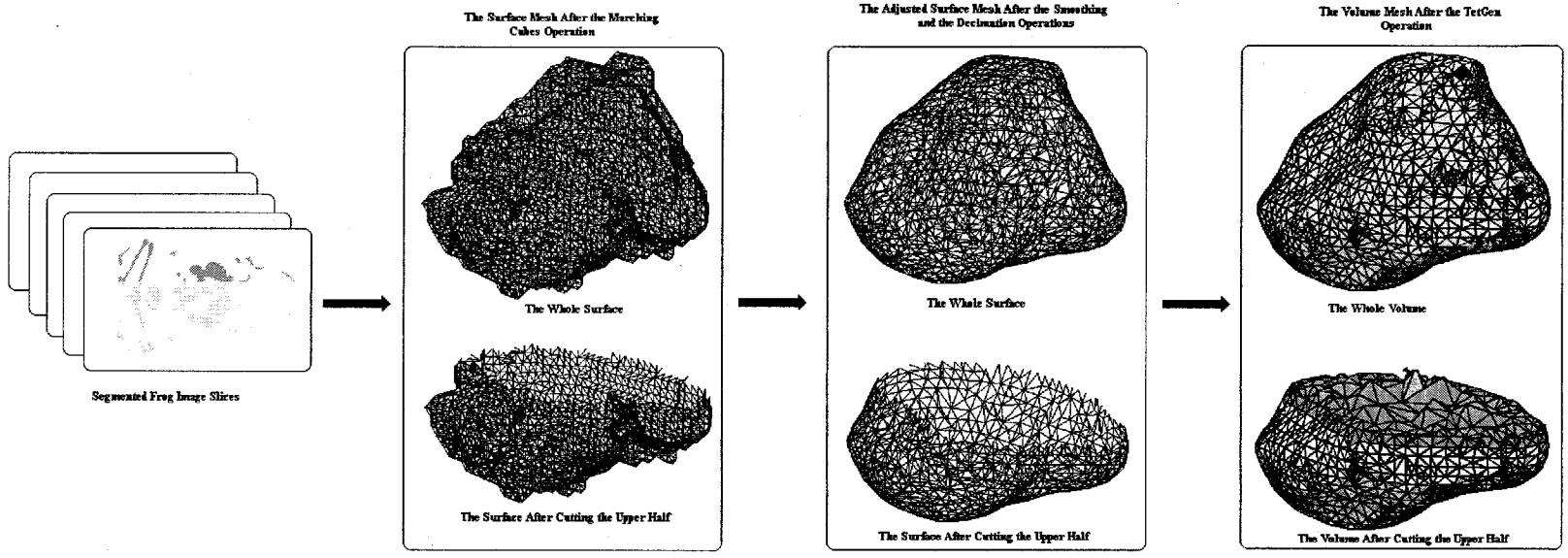

Figure 5.2: Flow chart for the creation of the frog's spleen volume mesh.

5.2 Effect of the Decimation Operation

The decimation operation is done by the vtkDecimatePro filter. This filter implements a very similar algorithm to the one described in section 2.6.2 except that it is designed to generate progressive meshes that is a stream of operations that can be easily transmitted and incrementally updated [14,40]. The experiments done with this operation is to evaluate the effect of changing the reduction rate parameter on the quality of the generated mesh and also to evaluate whether a different decimation approach would have any different results when applied to the same data.

5.2.1 Decimation Reduction Rate Effect

In this experiment, the same operations in Figure 5.1 were carried out multiple times. In each time, the reduction rate was chosen to be a value of 0, 0.2, 0.4, 0.6, 0.8, or 0.95. Similar tables to Tables 5.1 and 5.2 were produced and compared. Even though the numerical values of the geometric mesh quality measures were changing slightly between the different iterations of the experiments, there was no clear direction for these changes. This would suggest the low impact of the decimation operation on the geometric measures of the produced elements. On the other hand, there was a clear impact on the mesh performance measures. This can be seen in Figure 5.3 where the number of the tetrahedra generated gets changed in almost a regular fashion for all the tissues.

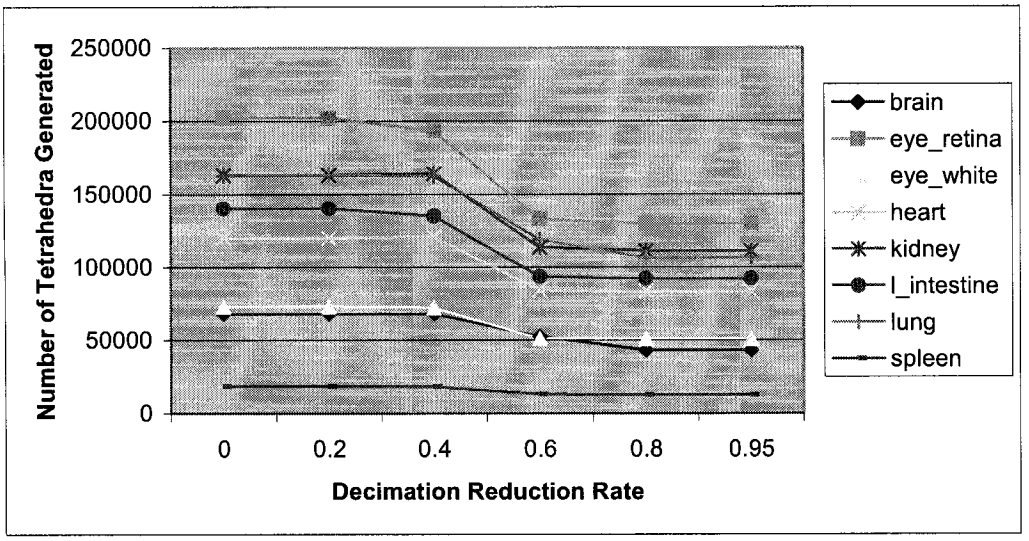

Figure 5.3: Effect of the decimation reduction rate on the number of generated elements.

The increase of the reduction rate at the low spectrum of the chart doesn't have a measurable effect on the number of generated elements. This would suggest that the small reduction in the number of the triangles on the surface mesh is not sufficient for TetGen to start producing large elements that conform to the 1.414 edge-ratio imposed constraint and therefore not able to decrease the number of generated elements. The situation changes dramatically once the reduction becomes relatively higher. TetGen would have higher freedom to generate larger elements and therefore reduce the number of generated elements. However, this doesn't last for long as the number of generated elements in the mesh becomes constant at the high spectrum of the chart. This can be explained better with the aid of Figure 5.4.

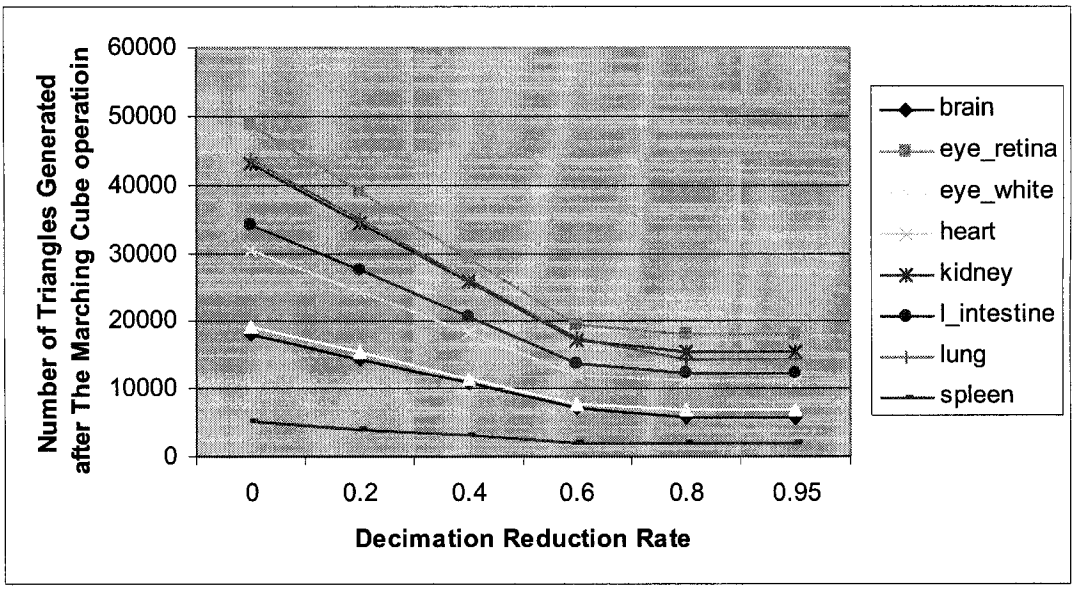

Figure 5.4: Effect of the decimation reduction rate on the number of triangles generated on the mesh surface.

In Figure 5.4, one can realize that there is no actual decrease in the number of the triangles on the mesh surface as one would expect with a higher reduction rate. The reason is because the vtkDecimatePro implementation doesn't guarantee satisfying the reduction rate [14]. It is a challenge for a decimation algorithm to obtain a very high reduction rate without changing significantly the topology of the mesh. vtkDecimatePro was used and instructed to preserve the topology of the mesh as much as possible.

Similar effect was reported on the various other performance mesh measures. For instance, Figure 5.5 shows how the average tetrahedron volume gets affected by the variation of the reduction rate.

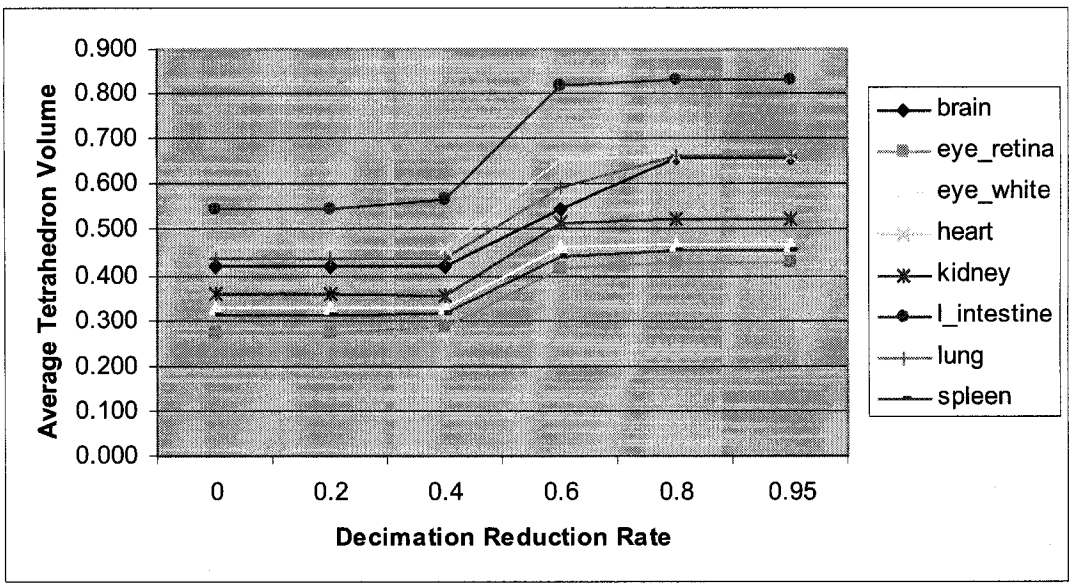

Figure 5.5: Effect of the decimation reduction rate on the average tetrahedron volume.

5.2.2 Different Decimation Implementation

In this experiment, the same operations in Figure 5.1 will be carried out again but this time with a different implementation for the decimation operation. More specifically, the vtkDecimatePro object will be replaced with the vtkQuadricDecimation object. The details of the vtkQuadricDecimation algorithm can be found in [41]. Figures 5.6 and 5.7 are the vtkQuadricDecimation figures corresponding to Figures 5.3 and 5.4. Similar behaviour was observed except for the high reduction rate spectrum. This change in behaviour would be explained by the ability of vtkQuadricDecimation to achieve higher reduction by changing significantly the topology of the mesh. This can be numerically seen in table 5.3 where the symmetrical Hausdorff distance was measured from the original surface produced by the Marching Cubes algorithm for the spleen tissue to the produced decimated surface by both implementations. Both implementations have relatively equal distance when the reduction rate is low. However; one can clearly see how the distance becomes large for the case of vtkQuadricDecimation implementation in the high reduction rate.

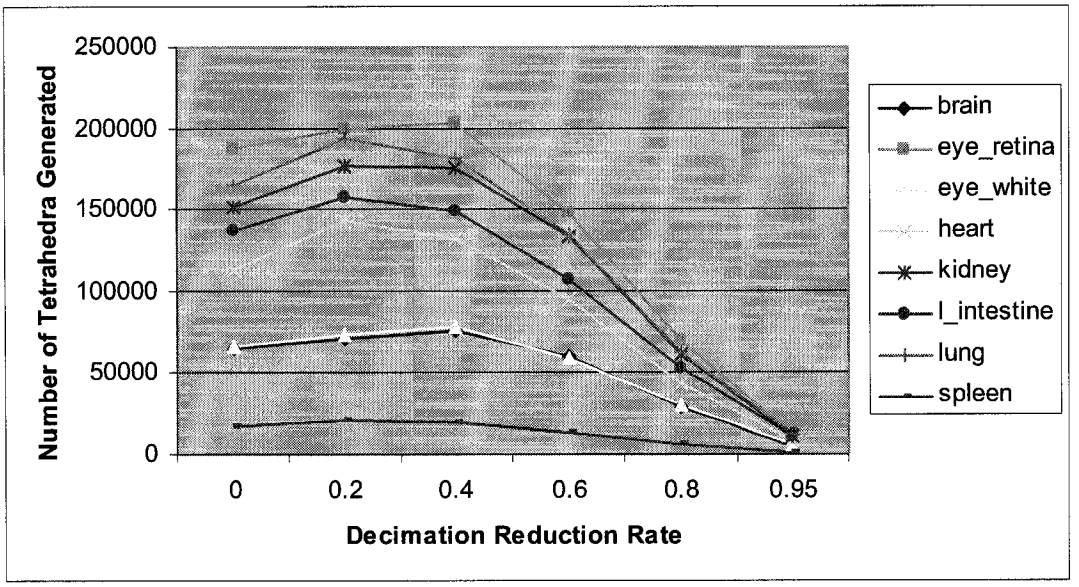

Figure 5.6: Effect of the vtkQuadricDecimation reduction rate on the number of tetrahedra generated.

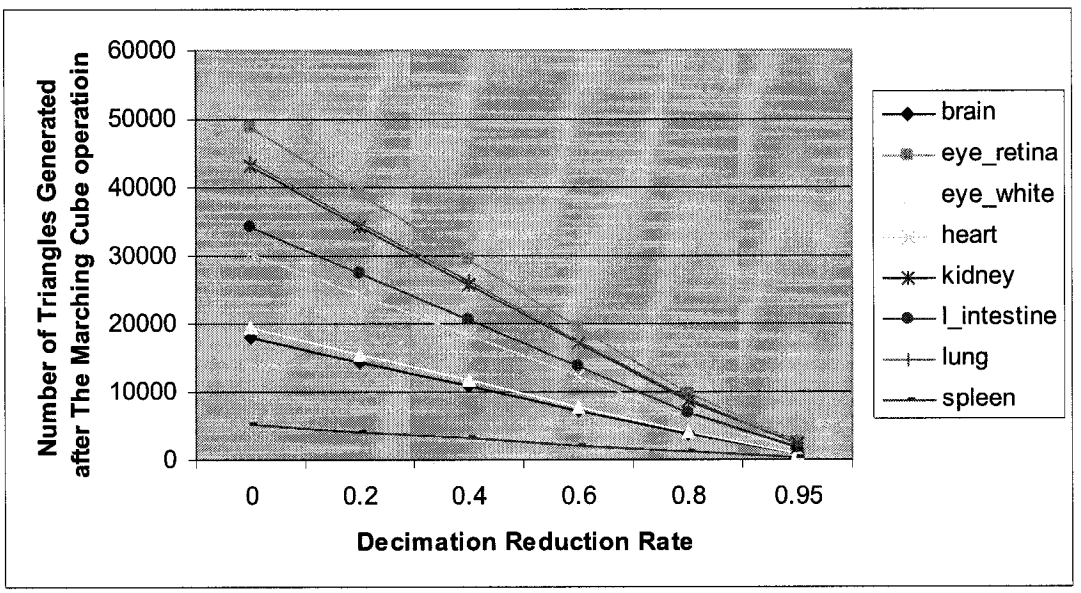

Figure 5.7: Effect of the vtkQuadricDecimation reduction rate on the number of triangles generated on the mesh surface.

Table 5.3: Symmetrical Hausdorff distance comparison between the surfaces produced by vtkDecimatePro against vtkQuadricDecimation.

	vtkDecimatePro			vtkQuadricDecimation		
	Symmetrical Hausdorff distance			Symmetrical Hausdorff distance		
Reduction Rate	Max	Mean	RMS	Max	Mean	RMS
0.02	0.378	0.013	0.024	0.253	0.012	0.025
0.04	0.430	0.023	0.035	0.335	0.025	0.038
0.06	0.507	0.036	0.048	0.363	0.039	0.052
0.08	0.507	0.037	0.051	39.893	0.220	1.168
0.95	0.507	0.037	0.051	40.751	0.686	1.422

5.3 Effect of the Smoother Operation

In this section, the experiments will be carried out to analyse the effect of the parameters of the smoother operation on the quality of the final mesh. There are mainly two parameters that control the smoother operations: (i) the weight factor used in the Laplacian equation; and (ii) the number of times the Laplacian equation is executed on the same vertex. A similar approach was followed here as in the previous section to study the effect of these parameters.

5.3.1 Effect of the Weight Factor of the Smoother Operation

In this experiment, the same operations in Figure 5.1 were carried out multiple times. In each time, the weight factor parameter of the smoother operation was set to a different value from the set {0.01, 0.03, 0.05, 0.07, 0.1}. The other parameters were set to the same values as in the reference case. On average and unlike the case of decimation reduction rate, there is a clear small positive change in the geometric mesh measures. The smoothing operation relaxes the meshes by making the triangles better shaped and the vertices more evenly distributed. The improved input set of vertices and triangles describing the object surface helped the TetGen mesh generator to produce tetrahedra

with better geometric quality. Figures 5.8 and 5.9 depict how the average minimal dihedral angles and the average β_{Baker} of the generated tetrahedra change as the weight factor of the smoother operation changes.

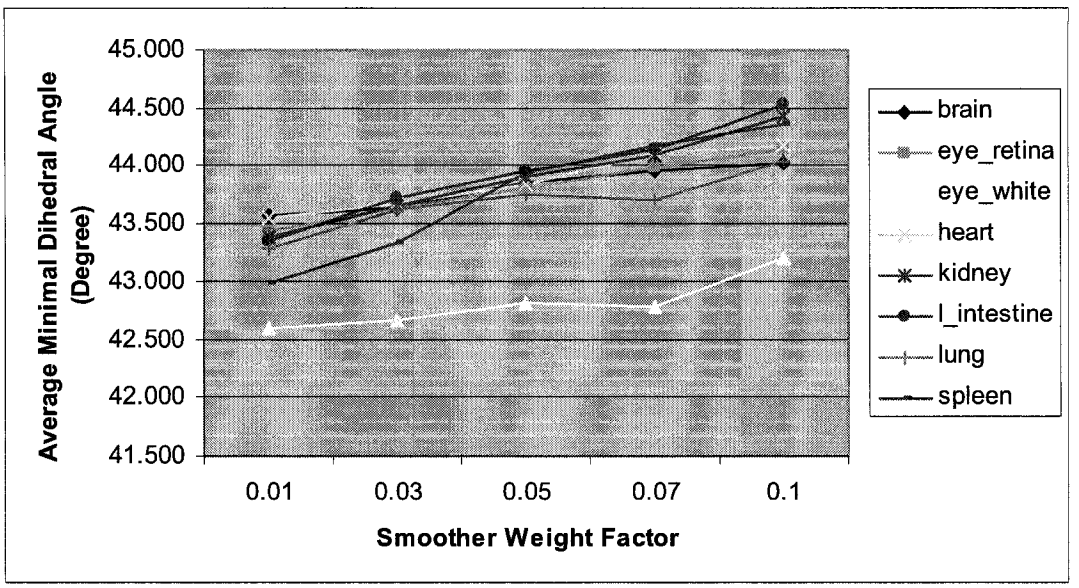

Figure 5.8: Effect of the weight factor of the smoother on the average minimal dihedral angle.

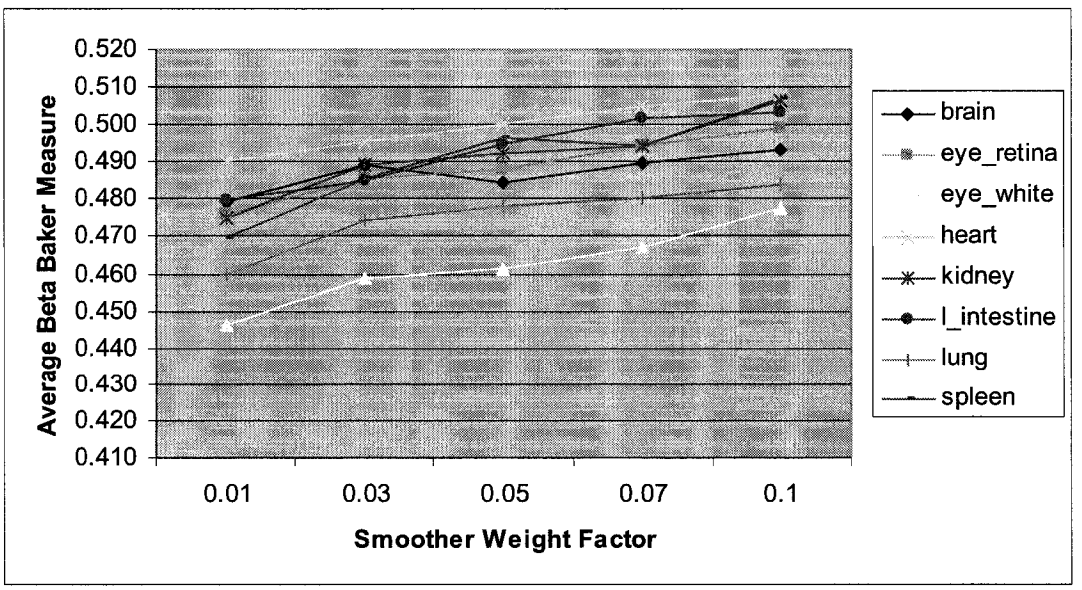

Figure 5.9: Effect of the weight factor of the smoother on the average β_{Baker}.

Also, the variation of the weight factor parameter shows a clear change in the performance measure. For example, the change in the number of generated tetrahedra can be seen in Figure 5.10. This change can be explained by two reasons: (i) the improved shaped triangles input gives TetGen a better opportunity to produce fewer tetrahedra with larger volumes that still comply to the specified input constraint (q1.414 switch), and (ii) the smoothing operation has a shrinking effect of the object being smoothed and therefore the volume of the object would be less as more smoothing is done.

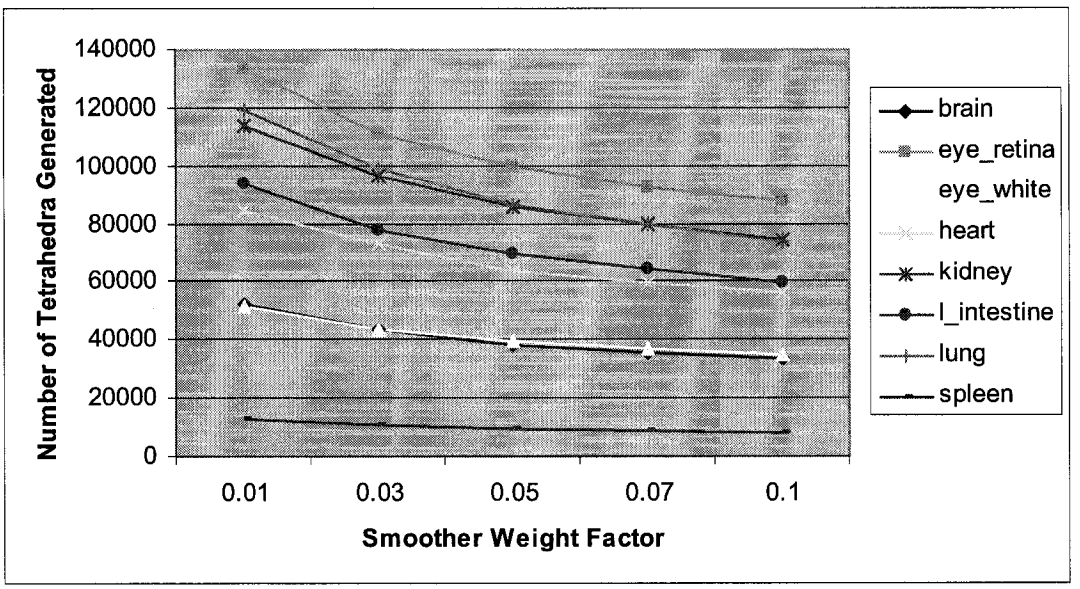

Figure 5.10: Effect of the weight factor of the smoother on the number of generated tetrahedra.

5.3.2 Effect of the Number of Iterations of the Smoother Operation

Similar experiments were carried out again but this time while varying the number of iterations parameter of the smoother operation. This parameter took a value from the set {10, 30, 50, 70, 100} in each experiment. Very similar results were obtained as the experiments done for the weight factor parameter. This would suggest that both

parameters would have similar effect on the quality of the final mesh. Figures 5.11 and 5.12 show some of the obtained results.

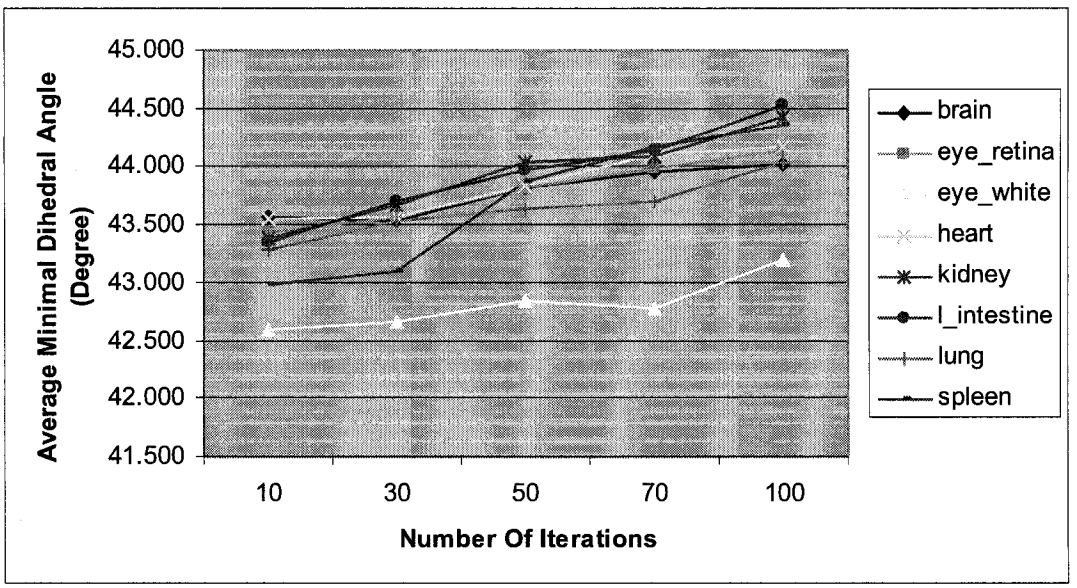

Figure 5.11: Effect of the number of iterations of the smoother on the average minimal dihedral angle.

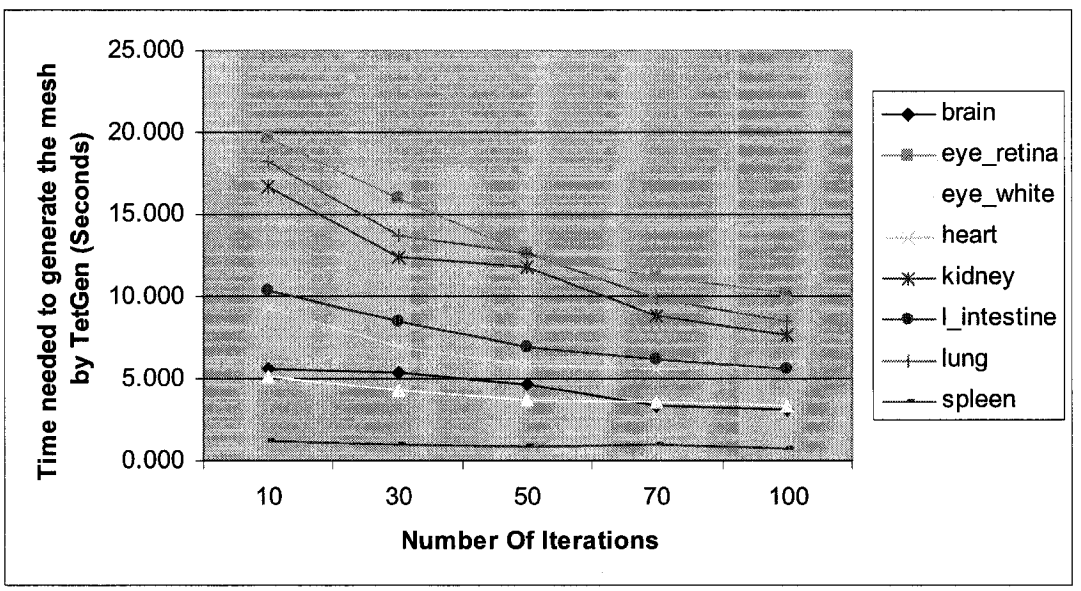

Figure 5.12: Effect of the number of iterations of the smoother on the time needed to generate the tetrahedra.

5.3.3 The Combined Effect of Both Parameters of the Smoother

Since both smoother parameters have a similar effect on the quality of the final mesh. One would expect to enforce the expected change if both parameters were to be used in the same experiment. Tables 5.4 and 5.5 show the obtained numerical results of the geometric and performance quality measures when repeating the experiment by setting the weight factor to .05 and the number of iterations to 50. Most of the time, It can be seen that it is possible to achieve at least the same effect as if the experiment was done by only setting the weight factor to 0.1 or by setting only the number of iterations parameter to 100.

It is worth noting that Laplacian smoothing works well in most cases but one has to remember that mesh smoothing modifies point's coordinates, and therefore, surface geometry. Excessive smoothing can badly damage the mesh. Large numbers of smoothing iterations, or large smoothing factors, can cause excessive shrinkage and surface distortion. Some object like sphere may lose volume and even shrink to a point [14].

Table 5.4: Numerical results for the geometric mesh quality measures when the smoother factor is set to 0.05 and the number of iteration is set to 50.

Tissue Name	Dihedral Angle			α			β_{Baker}		
	Min	Average	Max	Max	average	Min	Max	average	Min
brain	1.422	44.415	87.593	0.998	0.556	0.011	0.974	0.502	0.024
eye_retina	0.889	44.360	88.514	1.000	0.556	0.001	0.987	0.506	0.010
eye_white	1.498	43.635	87.412	0.999	0.528	0.012	0.981	0.483	0.015
kidney	1.894	44.569	88.016	0.998	0.570	0.009	0.976	0.509	0.010
l_Intestine	0.579	44.888	87.936	0.998	0.481	0.000	0.972	0.511	0.006
lung	0.850	44.286	88.021	0.999	0.521	0.001	0.985	0.496	0.005
spleen	5.668	45.035	85.304	0.991	0.586	0.022	0.958	0.519	0.088

Table 5.5: Numerical results for the performance mesh quality measures when the smoother factor is set to 0.05 and the number of iteration is set to 50.

Tissue Name	# of Tetrahedron generated	Tetrahedron Volume			Mesh Generation Time (s)	Memory (KB)
		Min	average	Max		
brain	31468	0.00025	0.857	34.901	3.128	3917
eye_retina	84392	0.00002	0.637	16.571	9.585	10544
eye_white	34324	0.00003	0.637	24.112	3.409	4323
kidney	73723	0.00001	0.764	20.126	7.741	9132
l_intestine	57397	0.00003	1.299	81.441	5.335	7157
lung	67821	0.00002	1.006	62.983	7.502	8809
spleen	7310	0.00003	0.716	13.845	0.605	981

5.4 Effect of Adding a New Operation

In this section, an attempt to introduce a new operation to the set of operations in Figure 5.1 will be discussed. This is a pre-processing operation that can aid in supplying different set of initial nodes to the Marching Cubes operation. The objective is to be able to generate better quality meshes with the minimum distortion of the surfaces of the scanned objects.

The idea is to smooth the volume data of the segmented images before creating the isosurface of the object. Noise is inherent in all methods of data acquisition including the MRI scanner. By introducing this new operation, the segmented images will be blurred and the noise should be reduced. This will be accomplished by making use of the vtkImageGaussianSmooth filter which implements a convolution of the input image with a Gaussian kernel. The amount of smoothing is controlled by two parameters: (i) the Gaussian standard deviation; and (ii) the radius of the kernel used. Two set of experiments were carried out to study the effect of the two parameters. The first set was done by varying the radius factor of the kernel from 1 to 4 in each direction simultaneously and keeping the standard deviation factor set to 1 in each direction. Table 5.6 reports the number of generated tetrahedra in the final mesh of each tissue. One can notice from the table that once the Gaussian smoother operation gets applied, a reduction in the number of generated elements occurs. However, this amount of reduction doesn't change as the radius factor increases and it is almost constant. This is mainly because the new nodes enclosed by the extended volume introduced by the increased radius value don't get enough weight from the Gaussian kernel to contribute enough to make a change of the current geometry. This can be seen clearly in Table 5.7

where the symmetrical Hausdorff distances are calculated from the different brain surfaces produced in the experiments to the brain surface produced when no Gaussian smoothing is applied. This distance is almost constant for the cases when the radius equals 2, 3, or 4.

On the other hand, there were no clear changes in the numerical results of the geometric mesh quality measures during the different experiments of this set.

Table 5.6: The effect of Gaussian smoothing radius factor on the number of tetrahedra generated.

	Number of Tetrahedra Generated				
Tissue Name	No Gaussian	R= (1,1,1)	R= (2,2,2)	R= (3,3,3)	R= (4,4,4)
brain	52515	42612	42441	41005	41420
eye_retina	133186	115741	115519	117039	117636
eye_white	51318	40049	36046	36495	35820
heart	84468	78372	79029	78281	80673
kidney	113752	107195	111192	112430	110386
l_intestine	93577	95676	96650	99514	98771
lung	119058	114645	114862	115384	114983
spleen	13012	12296	10890	10943	10978

Table 5.7: The effect of Gaussian smoothing radius factor on the symmetrical Hausdorff distance (brain tissue).

Gaussian Radius Factor	symmetrical Hausdorff distance		
	Max	Mean	RMS
(1,1,1)	3.570	0.349	0.501
(2,2,2)	4.794	0.446	0.671
(3,3,3)	4.795	0.453	0.677
(4,4,4)	4.795	0.453	0.678

The second set of experiments was carried out to evaluate the effect of the Gaussian standard deviation factor on the meshes generated. This was done by varying the standard deviation factor from 1 till 4 while keeping the radius of the kernel constant at 1. Figure 5.13 shows how the number of elements generated decreases with the increase of the standard deviation value. The increase in the value of the standard deviation gives

the adjacent nodes more weight to be able to contribute in changing the geometry of the surface. Table 5.8 shows how this is actually represented in numbers as the symmetrical Hausdorff distances between the surface of the un-smoothed brain tissue and the other generated surfaces become larger when the standard deviations increases in value.

Again, there were no clear changes in the numerical results of the geometric mesh quality measures during the different experiments of this set as well.

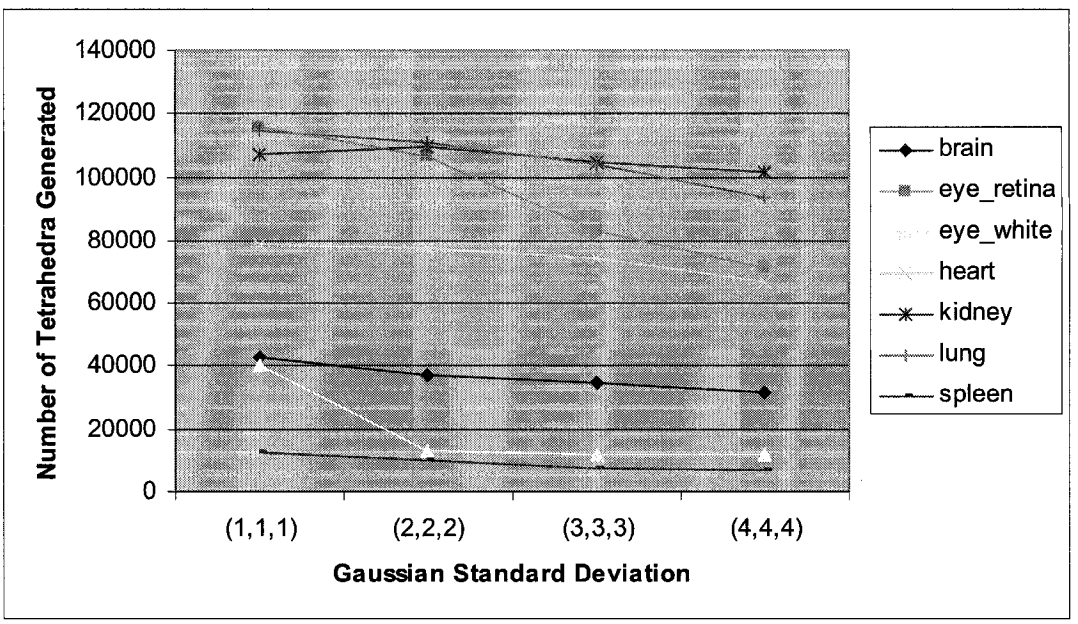

Figure 5.13: Effect of the Gaussian smoothing standard deviation factor on the number of generated tetrahedra.

Table 5.8: Effect of the Gaussian smoothing standard deviation factor on the symmetrical Hausdorff distance (brain tissue).

Gaussian Standard deviation	symmetrical Hausdorff distance		
	Max	Mean	RMS
(1,1,1)	3.570	0.349	0.501
(2,2,2)	6.859	0.690	1.040
(3,3,3)	9.343	1.057	1.645
(4,4,4)	10.286	1.399	2.073

Finally, it is worth reporting that some other attempts were made to introduce different operations in the proposed process. For example, one of these attempts was to introduce a sub-sampling operation. Usually the data obtained from the MRI scanner are of high resolution. The idea was to sub-sample the segmented volume data to a lower resolution volume that would eventually produce fewer triangles. The experiments were carried out to study the effect of such operation on the quality of the produced meshes. It was shown that the sub-sampling operation led to fewer elements produced but with relatively large symmetrical Hausdorff distances. This is also beside the fact that the geometric quality measures became worse. These are mainly because important details were lost during the sub-sampling operation.

Chapter 6

Conclusion and Future work

6.1 Future Work

A very interesting possible future work would be to extend the current design and implementation to support parallelism. For example, recent work in VTK showed that it is possible to utilize this individual software component separately to provide a parallel system solution that is scalable and portable [46]. This; however, needs to be enhanced with the other software components. The ITK component supports shared-memory parallel processing but not distributed-memory parallel processing. TetGen doesn't support parallelism and therefore it is possible to be modified or to be replaced with a different software component. One possible approach would be to use a mesh generator that relies on the advancing front techniques which doesn't introduce any new points on the original surface mesh and therefore makes it perfect for parallelism by splitting the original object domain in multiple sub domains [17].

Another possibility for future work would be to experiment with non-tetrahedral mesh generating algorithms. For instance, hexahedral meshes are well suited for the deformation computation of incompressible materials [24]. This type of meshes, however; needs more sophisticated hierarchical adaptive refinement of basis functions for achieving similar results compared to the tetrahedral straightforward manner [24].

6.2 Conclusion

This thesis described the process to produce three dimensional mesh models from a sequence of MRI medical images. This process can be summarized in four major steps. The first step is the segmentation. The second step is generating the surface boundary

mesh from the segmented images. The third step is to adjust the surface boundary. The final step is to create the volume meshes from the adjusted surface meshes generated earlier. At the end, the quality of the mesh generated depends on the various algorithms and techniques used during these four steps. A study on how some of these low level operations might affect the quality of the produced meshes was presented and discussed. A usable, open-source, portable, efficient, and extendible system architecture was proposed to accomplish the task of generating meshes from the native medical file format.

In conclusion, the quality of the final meshes generated can be enhanced throughout the various steps needed to generate them.

References

[1] D. Giannacopoulos and S. McFee, "Functional Derivatives and Optimal Discretization Based Refinement Criteria for Adaptive Finite Element Analysis with Scalar Tetrahedra," IEEE Transactions on Magnetics, 1999, pp. 1326-1329.

[2] M. Dorica and D. Giannacopoulos, "Towards Optimal Mesh Quality Improvements for Adaptive Finite Element Electromagnetics with Tetrahedra," IEEE Transactions on Magnetics, 2004, 40(2).

[3] Claes Lundström, *Segmentation of medical image volumes*. Master thesis, Linköping University, November 1997.

[4] W. Lorensen and H. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," Computer Graphics, 21 (4): 163-169, July 1987.

[5] Matthieu Ferrant, Simon K. Warfield, Arya Nabavi, Ferenc A. Jolesz, and Ron Kikinis, "Registration of 3D Intraoperative MR Images of the Brain Using a Finite Element Biomechanical Model," Source Lecture Notes In Computer Science, Proceedings of the Third International Conference on Medical Image Computing and Computer-Assisted Intervention, Vol. 1935, 2000, pp. 19-28.

[6] Timoner S., *Compact Representations for Fast Nonrigid Registration of Medical Images*. PhD thesis, MIT, 2003.

[7] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen, "Decimation of Triangle Meshes," International Conference on Computer Graphics and Interactive Techniques, Proceedings of the 19th annual conference on computer graphics and interactive, 1992, pp. 65-70.

[8] John M. Sullivan, Jr., Ziji Wu, and Anand Kulkarni, "3D Volume Mesh Generation of Human Organs Using Surface Geometries Created from the Visible Human Data Set," The Third Visible Human Project Conference Proceedings, Bethesda, Maryland, October 5-6, 2000.

[9] John Melonakos, Ramsey Al-Hakim, and James Fallon, "Knowledge-Based Segmentation of Brain MRI Scans Using the Insight Toolkit," IJ - 2005 MICCAI Open-Source Workshop, Oct-2005.

[10] Andriy Fedorov, Nikos Chrisochoides, Ron Kikinis, and Simon K. Warfield, "Tetrahedral Mesh Generation for Medical Imaging," 2005 MICCAI Open Source Workshop, 2005.

[11] A. Mohamed and C. Davatzikos, "Finite Element Mesh Generation and Remeshing from Segmented Medical Images," IEEE International Symposium on Volume, Vol 1, 15-18 April 2004, pp. 420– 423.

[12] Jonathan Richard Shewchuk, *Lecture Notes on Delaunay Mesh Generation*, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA, 1999.

[13] Xenophon Papademtris, *An Introduction to Programming for medical image Analysis with the visualization Toolkit,* Yale University, 2006.

[14] Will Schroeder, Ken Martin, and Bill Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Upper Saddle River, NJ: Prentice Hall, 1998.

[15] L. Ibanez and W. Schroeder, *The ITK Software Guide*, Kitware, Inc. ISBN 1-930934-10-6, 2003, http://www.itk.org/ItkSoftwareGuide.pdf.

[16] Du Jian-jun, Yang Xiao-yong, and Du You-jun, "From Medical Images to Finite Grids System," IEEE-EMBS 2005. 27th Annual International Conference of the Volume, 01-04 Sept. 2005, pp.1630-1633.

[17] A. Fedorov, N. Chrisochoides, R. Kikinis, and S. K. Warfield, "An Evaluation of Three Approaches to Tetrahedral Mesh Generation for Deformable Registration of Brain MR Images," Biomedical Imaging: Nano to Macro,3rd IEEE International Symposium on Volume, 6-9 April 2006, pp. 658-661.

[18] A. Coronato, G. De Pietro, and I. Marra, "An Open-source Software Architecture for Immersive Medical Imaging," Virtual Environments, Human-Computer Interfaces and Measurement Systems, Proceedings of 2006 IEEE International Conference. July 2006, pp. 166-170.

[19] N. Aspert, D. Santa-Cruz, and T. Ebrahimi, "MESH: Measuring Errors Between Surfaces Using the Hausdorff Distance," IEEE International Conference on Vol. 1, 2002, pp. 705-708.

[20] N. Archip, and R. Rohling, "Volumetric anatomical modeling from medical images," Engineering in Medicine and Biology Society, 26th Annual International Conference of the IEEE, Vol.3, 1-5 Sept. 2004, pp. 1838-1841.

[21] J.R Shewchuk, "What is a Good Linear Element? Interpolation, Conditioning and Quality Measures," Proceedings of the 11th International Meshing Roundtable (Sandia National Laboratories, 15-18 Sep 2002, pp. 115-126.

[22] P.G. Ciarlet, "The Finite Element Method for ELLIPTIC Problems," North Holland, Amsterdam, 1978.

[23] I. Tsukerman, "Approximation of Conservative Fields and the Element Edge Shape Matrix," IEEE Transactions on Magnatics, 1998, pp. 34(5): 3248-3251

[24] M. Sermesant, C. Forest, X. Pennec, H. Delingette, and N. Ayache, "Biomechanical Model Construction from Different Modalities: Application to Cardiac Images," Lecture Notes in Computer Science, Medical Image Computing and Computer-Assisted Intervention - MICCAI 2002: 5th International Conference, Tokyo, Japan, September 25-28, 2002, Proceedings, Part I. pp. 714-721.

[25] Mark Dorica, *Novel Mesh Quality Improvement Systems for Enhanced Accuracy and Efficiency of Adaptive Finite Element Electromagnetic with Tetrahedra.* Master thesis, McGill University, April, 2004.

[26] T.J. Baker and J.C Vassberg, "Tetrahedral Mesh Generation and Optimization," Proceedings of the 6th International Conference on Numerical Grid Generation in Computational Field Simulations (University of Greenwich), July 1998, pp. 337-349.

[27] P.R. Cavalcanti and U.T. Mello, "Three-Dimensional Constrained Delaunay Triangulatio: A Minimalist Approach," Proceedings of the 8th International Meshing Roundtable (South Lake Tahoe, CA, U.S.A) Oct 1999, pp. 119-129.

[28] John E. Stewart, James H. Johnson, and William C. Broaddus, "Segmentation and Reconstruction Strategies for the Visible Man," Proceedings of the Visible Human Project Conference; 1996 Oct 7-8; Bethesda.

[29] N. P. Weatherill and O. Hassan, "Efficient Three-dimensional Delaunay Triangulation with Automatic Point Creation and Imposed Boundary Constraints", International Journal for Numerical Methods in Engineering, 1994, vol 37, pp.2005-2039

[30] I. Tskerman and A. Plaks, "Comparison of Accuracy Criteria for Approximation of Conservative Fiels in Tetrahedra," IEEE Transactions on Magnatics, 1998, pp. 34(5): 3252-3255.

[31] Jonathan Richard Shewchuk, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator", 1996, http://www.cs.cmu.edu/~quake/triangle.html

[32] S. Rebay, "Efficient Unstructured Mesh Generation by Means of Delaunay Triangulation and Bowyer-Watson Algorithm", 1993, Journal of Computational Physics, vol. 106, pp.125-138

[33] V.N. Parthasarathy, "A Comparison of Tetrahedron Quality Measures," Finite Elements in Analysis and Design, 1993, 15:255-261.

[34] D. Field, "Qualitative Measures for Initial Meshes," International Journal for Numerical Methods in Engineering, 2000, 47: 887-906.

[35] T.J. Baker and J.C. Vassberg, "Element Quality in Tetrahedral Meshes," Proceedings of the 7th International Conference on Finite Element Methods in Flow Problems, Numerical Grid Generation in Computational Field Simulations (Huntsville, U.S.A), 1989, pp1018-1024.

[36] T.S. Yoo, M. J. Ackerman, W. E. Lorensen, W. Schroeder, V. Chalana, S. Aylward, D. Metaxes, and R. Whitaker, "Engineering and Algorithm Design for an Image Processing API: A Technical Report on ITK - The Insight Toolkit," In Proc. of Medicine Meets Virtual Reality, J. Westwood, ed., IOS Press Amsterdam pp 586-592 (2002).

[37] P.L. George, F. Hecht and E. Saltel, "Automatic Mesh Generator with Specified Boundary," Computer Methods in Applied Mechanics and Engineering, North-Holland, 1991, vol 92, pp.269-288

[38] Jonathan Richard Shewchuk, *Delaunay Refinement Mesh Generation.* Ph.D. thesis, Carnegie-Mellon Univ., School of Computer Science, May 1997

[39] H. Borouchaki, F. Hecht, E. Saltel and P. L. George, "Reasonably Efficient Delaunay Based Mesh Generator in 3 Dimensions," Proceedings 4th International Meshing Roundtable, pp.3-14, October 1995

[40] Hugues Hoppe, "Progressive meshes," In SIGGRAPH 96 Conference Proceedings, pages 99-108. ACM SIGGRAPH, Addison Wesley, August 1996.

[41] Hugues Hoppe, "New quadric metric for simplifying meshes with appearance attributes," IEEE Visualization 1999, October 1999, pp. 59-66.

[42] The whole frog project, Lawrence Berkeley National Laboratory, http://froggy.lbl.gov/

[43] The Visualization ToolKit (VTK) website, http://www.vtk.org/

[44] NLM Insight Segmentation and registration Toolkit website, http://www.itk.org/

[45] The TetGen project, Numerical Mathematics and Scientific Computing research group, Weierstrass Institute for Applied Analysis and Stochastics, http://tetgen.berlios.de/

[46] James Ahrens, Charles Law, Will Schroeder, Ken Martin, Michael Papka, "A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets" Tech. Rep. LAUR-00-1620, Los Alamos National Laboratory.

[47] Tcl Developer Xchange website, http://www.tcl.tk/

[48] Blair Mackiewich, *Intracranial boundary detection and radio frequency correction in magnetic resonance images,* Master thesis, Simon Fraser University, August 1995.

[49] A. Glowinski, J. Kursch, G. Adam, A. Bucker, T.G. Noll, and R.W. Gunther, "Device visualization for interventional MRI using local magneticfields: basic theory and its application to catheter visualization," IEEE Trans Med Imag., vol. 17, pp. 786–793, Oct. 1998.

[50] Steve Owen, "A survey of unstructured mesh generation technology," in Proceedings of 7th International Meshing Roundtable, 1998, pp. 239–267

[51] National Electrical Manufacturers Association (NEMA), http://dicom.nema.org/

Appendix I: Detailed Instructions on How to Setup the Proposed System

In this appendix a summary of the instructions needed to setup the proposed system environment and to connect the different software components with each other is presented. These instructions are provided for the MS Windows platform. The steps are very similar in any different platform.

Required Tools:

TCL/TK:

Tcl scripting language comes preinstalled on most UNIX systems and Mac OS X; however, one needs to download and install the ActiveTcl distribution from www.tcl.tk for a Windows platform.

Cross-Platform Make (CMake):

CMake can be downloaded/installed from www.cmake.org

C++ Compiler:

Because of the use of the CMake tool, any standard C++ compiler in any platform can be used. MS Visual Studio 6 SP6 was chosen for the various experiments done in this thesis.

Optional Tools:

CVS:

CVS is an open-source version control system that can keep a history record of the files. CVS repository is widely used by many developers to store/share both the code and the documentation of a project. In fact, many of the software components used in the proposed system can be optionally obtained from public

access CVS repositories. These software components are still under ongoing development. By using CVS, one would be able to take advantage of the changes added to these components more often without the need to repeat the download and install process with every new release. In order to be able to use CVS in Windows, one needs to download/install CVSNT from www.cvsnt.org. CVSNT provides its users with the CVS capabilities through the command line switches. A more user-friendly application with graphical interface can be installed on top of CVSNT. WinCvs from www.wincvs.org is one example of such GUI CVS front-end.

Instructions on how to setup the Visualization Toolkit (VTK):

1. Download the source code from www.vtk.org or
Checkout the VTK module from the CVS repository:
`:pserver:anonymous@public.kitware.com:/cvsroot/VTK`

Note that you might need the CVS source code to take advantage of the new functionalities provided by the modified vtkMeshQuality object.

2. Execute Cmake.exe
- In the "Where is the source code?" field, specify the location where the VTK source code was downloaded/checked out. This directory should contain a file called CMakeLists.txt

- In the "Where to build the binaries" field, specify the location where you would like to save the generated compiler/platform specific files. This directory is better to be empty as it will be the directory where all the binary libraries and executables will be generated for the project.

- Click on the Configure button.

- Choose the C++ compiler environment form the drop down list box.

- Wait till a list of cached values setting gets displayed. Change the VTK_WRAP_TCL setting to be ON and Click again on the configure button (See Figure 7.1) and wait till the OK button becomes enabled.

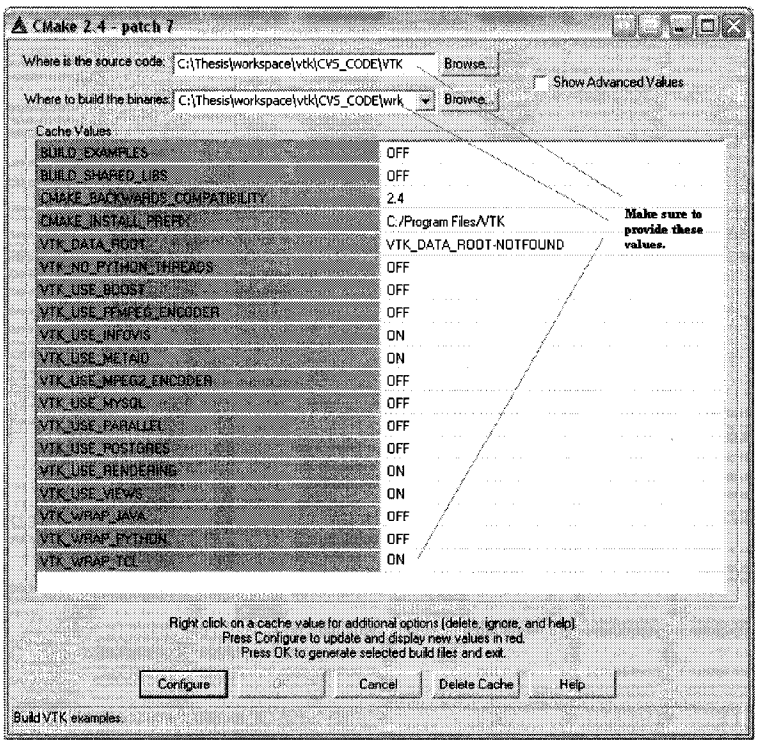

Figure 7.1: The Visualization Toolkit CMake Settings.

- Click on the OK button. CMake will start generating the needed files for your development environment and exit when it finishes.

3. Compile/Build the VTK code

Run the C++ development environment chosen in step#2 with the CMake generated file located in the root of build binaries directory chosen also in step#2. Unless there is a need to debug the code, building the release configuration should be sufficient and it should reduce the time needed to compile/build the code. It is worth noting that the compilation of the VTK library is a lengthy process and it might take a couple of hours.

Instructions on how to setup the TetGen Mesh Generator:

1. Download the source code from http://tetgen.berlios.de/ and extract the files to a local directory.

2. Add TclTetGen.cpp and CMakeLists.txt files to the same location where the source files were extracted (See Appendices II and III).

3. Execute Cmake.exe

- Specify the path to the TetGen source code directory in the "Where is the source code?" field, and a path to an empty directory in the "Where to build the binaries" field.

- Click on the Configure button and choose the C++ compiler environment form the drop down list box.

- Click again on the Configure button and then Click on the OK button when it gets enabled. At the end of this step the development environment files will be generated.

4. Compile/Build the TetGen code

Compile and Build the TetGen Code from the C++ compiler environment generated from step#3. Similarly to VTK, building the release configuration should be sufficient unless there is a need to debug the code. The time needed to compile/build the TetGen code is in the order of minutes.

Instructions on how to setup the Insight Toolkit (ITK):

1. Download the source code from www.itk.org or
Checkout the `Insight` module from the CVS repository:
`:pserver:anonymous@www.itk.org:/cvsroot/Insight`

2. Download the source code of CableSwig from www.itk.org or
Check-out the CableSwig module from the CVS repository:
`:pserver:anonymous@public.kitware.com:/cvsroot/CableSwig`

CableSwig is needed to create the Tcl wrappers of the functions of the ITK library. The downloaded CableSwig directory should be placed under the Utilities directory of the ITK Root directory. If the downloaded/extracted directory is not named CableSwig, Rename it to be CableSwig. At the end, the directory hierarchy should be as follows:
{Path to where ITK was extracted}\Insight\Utilities\CableSwig
Where the CableSwig directory would contain a file called CMakeLists.txt

3. Execute Cmake.exe
- In the "Where is the source code?" field, specify the location where the ITK source code was downloaded/checked out. This directory should contain a file called CMakeLists.txt. In the "Where to build the binaries" field, specify the location where to save the generated compiler/platform specific files.

- Click on the Configure button and choose the C++ compiler environment form the drop down list box when prompted.

- When the list of cached values setting gets displayed, check the Show Advanced values option and then modify the ITK_CSWIG_TCL setting to be ON (See Figure 7.2).

- Click again on the configure button and wait till the OK button becomes enabled.

- Click on the OK button. CMake will start generating the needed files for your development environment and exit when it finishes.

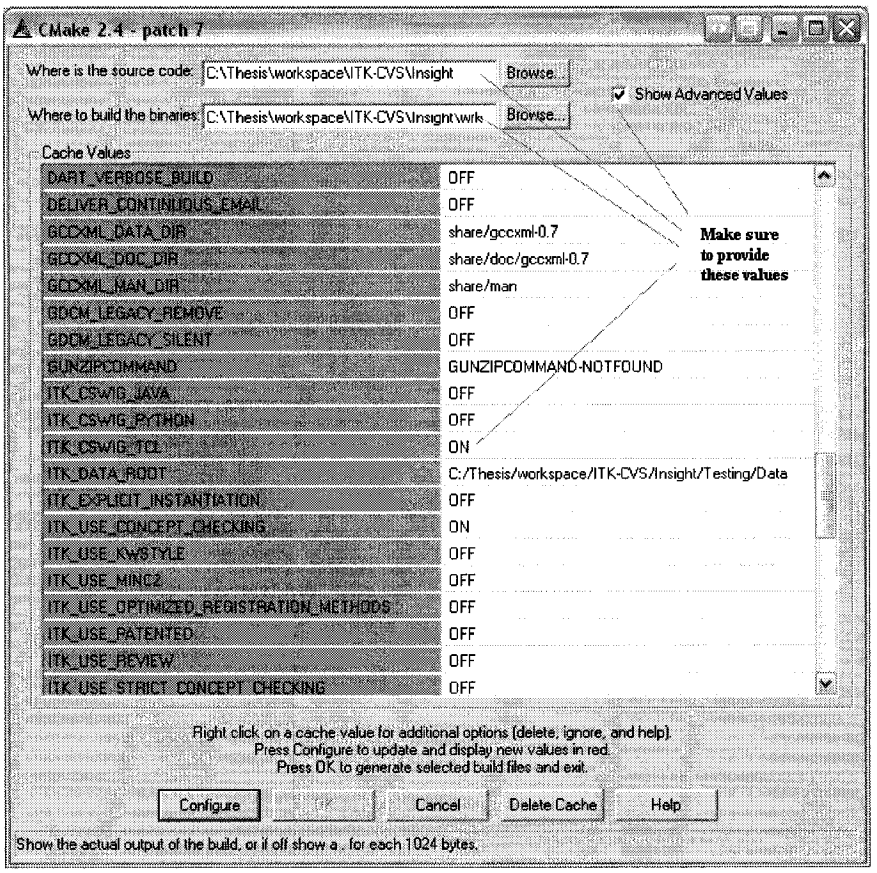

Figure 7.2: The Insight Toolkit CMake Settings.

3. Compile/Build the ITK code

Compile/Build the release configuration of the ITK code in the same way as was done for VTK. The compilation process of the ITK library is a lengthy process and it might take a couple of hours.

Instructions on how put it all together:

1. Create an empty directory in the local system. This directory will be used as a workspace to combine all the generated components.

2. Copy the following items to this workspace directory:
   - The VTK executable file (vtk.exe in Windows) from:
   {Path to the location where VTK executables got generated}\bin\Release

-The generated TclTetGen shared library (TclTetGen.dll in Windows) from
{Path to the location where TetGen source code }\Release

3. Add a new environment variable called TCLLIBPATH that contains the path:
{Path to where the ITK libraries got generated}/ /Wrapping/CSwig/Tcl/Release
Append this new variable to the PATH environment variable.

By executing the VTK executable, a Tcl Shell will start from which one can make use of any the functionalities of the combined components ITK, VTK, or TetGen. Tcl scripts can be written with any text editor and be loaded with the source command. For example, to execute the code in Appendix VIII, one can execute the following from the Tcl shell prompt:

source experiment.tcl

Appendix II: TetGen CMake Configuration File

```
# CMakeLists.txt CMake Configuration File
# Project: TclTetGen
# Objective: Provides a means for the TetGen code to be easily portable across
#          the different platforms

PROJECT(TCLTETGEN)

# Define the C++ Files that will go into the TetGen.lib
SET (TETGEN_SRCS
tetgen.cxx
predicates.cxx
tetgen.h
)

# Define the C++ Files that will go into the TclTetGen.dll
SET (TCLTETGEN_SRCS
tetgen.h
TclTetGen.cpp
)

# Set the default location for outputting the library
SET (LIBRARY_OUTPUT_PATH ${TCLTETGEN_SOURCE_DIR})

# Build the tetgen.lib
ADD_LIBRARY(TetGen STATIC ${TETGEN_SRCS})
ADD_DEFINITIONS(-DTETLIBRARY)

# Look for tcl library
# Try first with the predefined CMake Macro
SET (TETGEN_TCL_CANBUILD 1)
INCLUDE (${CMAKE_ROOT}/Modules/FindTCL.cmake)

IF(TCL_LIBRARY)
  # Succeeded with the CMake Macro to find TCL
  INCLUDE_DIRECTORIES(${TCL_INCLUDE_PATH})
ELSE (TCL_LIBRARY)
   # Failed with the CMake Macro to find TCL
   #Another possible approach to find TCL is to search in the PATH Environment
        FIND_PATH(TCL_INCLUDE_PATH tcl.h PATHS)
        IF (TCL_INCLUDE_PATH)
          INCLUDE_DIRECTORIES(${TCL_INCLUDE_PATH})
          FIND_LIBRARY(TCL_LIBRARY NAMES tcl tcl84 tcl8.4)
        ELSE (TCL_INCLUDE_PATH)
    # Set The failure flag because TCL Cannot be found on this system
         SET (TETGEN_TCL_CANBUILD 0)
        ENDIF (TCL_INCLUDE_PATH)
ENDIF(TCL_LIBRARY)
```

```
# Build the Tcltetgen.lib
IF(TETGEN_TCL_CANBUILD)
        ADD_LIBRARY(TclTetGen SHARED ${TCLTETGEN_SRCS})
        TARGET_LINK_LIBRARIES(TclTetGen ${TCL_LIBRARY} TetGen)
ENDIF(TETGEN_TCL_CANBUILD)
```

Appendix III: TclTetGen C++ Code

```
/* File Name: TclTetGen.cpp
  Objective: This code is the wrapper needed for the TetGen code to provide a
  TCL interface. This would allow us to leverage the full functionalities of TetGen
  with simple TCL commands

  Example: In a Tcl shell one can execute the following:

 TetGen pq1.414 c:\\Data\\mesh_surface.ply
  */

#include "tcl.h"
#include "tclDecls.h"
#include <stdio.h>
#include <string.h>
#include "tetgen.h"

// Initialization Code
#ifdef _WIN32
#define TETGENTCL_EXPORT __declspec( dllexport )
#else
#define TETGENTCL_EXPORT
#endif

#define COMMAND_PARAM_NUM 3 // Expected number of parameters
#define BUFFER_SIZE 1024    // Size of buffer to be used to store
                            // each Parameter

int TetGen(ClientData, Tcl_Interp* interp,int objc, Tcl_Obj *CONST objv[])
{
        // Validate the command line parameters
  if (objc != COMMAND_PARAM_NUM)
  {
          printf("Error in calling the TCL TetGen code.");
          printf("Usage: TetGen {TetGenSwitches} {PathToMeshFile}");
  }
  else
  {
        // Handling of command line arguments
        char szExecutableName[BUFFER_SIZE] = {0};
        char szSwitchCmd[BUFFER_SIZE]      = {0};
        char szMeshPathCmd[BUFFER_SIZE]    = {0};

        char* arrayCommanLine[COMMAND_PARAM_NUM];

        strcpy(szExecutableName, Tcl_GetString(objv[0]));
        strcpy(szSwitchCmd, Tcl_GetString(objv[1]));
        strcpy(szMeshPathCmd, Tcl_GetString(objv[2]));

        int nOverHeadSize = 5;// Add some extra space in the memory to avoid overflow
```

```
        int nArgSize = strlen(szExecutableName);

        if (nArgSize>0)
        {
                nArgSize += nOverHeadSize;
                arrayCommanLine[0] = new char[nArgSize];
                memset(arrayCommanLine[0], 0, nArgSize);

                strcpy(arrayCommanLine[0], szExecutableName);
        }

        nArgSize = strlen(szSwitchCmd);
        if (nArgSize>0)
        {
                nArgSize += nOverHeadSize;
                arrayCommanLine[1] = new char[nArgSize];
                memset(arrayCommanLine[1], 0, nArgSize);

                // Check if the switch argument has the -, add it if missing
                if (szSwitchCmd[0] != '-')
                {
                        strcpy(arrayCommanLine[1], "-");
                        strcat(arrayCommanLine[1], szSwitchCmd);
                }
                else
                {
                        strcpy(arrayCommanLine[1], szSwitchCmd);
                }
        }

        nArgSize = strlen(szMeshPathCmd);
        if (nArgSize>0)
        {
                nArgSize += nOverHeadSize;
                arrayCommanLine[2] = new char[nArgSize];
                memset(arrayCommanLine[2], 0, nArgSize);
                strcpy(arrayCommanLine[2], szMeshPathCmd);
        }

        // Initializing TetGen In/Out Structures
        tetgenbehavior tetgenBehavior;
        tetgenio in, addin, bgmin;

        bool bContinue = true;

        if(!tetgenBehavior.parse_commandline(COMMAND_PARAM_NUM,
arrayCommanLine)) {
                printf("Error in calling the parse_commandlin.");
                bContinue = false;
        }
```

```
        if (bContinue)
        {
                if (tetgenBehavior.refine) {
                        if (!in.load_tetmesh(tetgenBehavior.infilename)) {
                          printf("Error in calling the load_tetmesh.");
                          bContinue = false;
                        }
                } else {
                        if(!in.load_plc(tetgenBehavior.infilename,(int)
tetgenBehavior.object)) {
                          printf("Error in calling the load_plc.");
                          bContinue = false;
                        }
                }
        }

        if (bContinue)
        {
          if (tetgenBehavior.insertaddpoints) {
                        if (!addin.load_node(tetgenBehavior.addinfilename)) {
                          addin.numberofpoints = 0l;
                        }
                }

                if (tetgenBehavior.metric) {
                        if (!bgmin.load_tetmesh(tetgenBehavior.bgmeshfilename)) {
                          bgmin.numberoftetrahedra = 0l;
                        }
                }

                // Calling the TetGen code
                if (bgmin.numberoftetrahedra > 0l) {
                        tetrahedralize(&tetgenBehavior, &in, NULL, &addin, &bgmin);
                } else {
                        tetrahedralize(&tetgenBehavior, &in, NULL, &addin, NULL);
                }
        }

        if (arrayCommanLine[0])
        {
                delete[] arrayCommanLine[0];
        }

        if (arrayCommanLine[1])
        {
                delete[] arrayCommanLine[1];
        }

        if (arrayCommanLine[2])
        {
                delete[] arrayCommanLine[2];
```

```
            }
        }


    return TCL_OK;
}

/*
  C++ to Tcl Interface
*/

extern "C" {
int  TETGENTCL_EXPORT Tcltetgen_Init(Tcl_Interp* interp);
int  TETGENTCL_EXPORT Tcltetgen_SafeInit(Tcl_Interp* interp);
}

int Tcltetgen_Init(Tcl_Interp* interp)
{
  Tcl_CreateObjCommand(interp, "TetGen", TetGen,(ClientData) NULL,
(Tcl_CmdDeleteProc *) NULL);
  return TCL_OK;
}

int Tcltetgen_SafeInit(Tcl_Interp* interp)
{
  return Tcltetgen_Init(interp);
}
```

Appendix IV: Template File to Wrap C++ Code as a Tcl Command

```
/*
  File Name: MyFunction.cpp
Objective: This is a C++ template file that can be used by researches to wrap their C++
code to be used as Tcl commands which can then be invoked from Tcl scripts. This is
needed to create a dynamic library(.so of .dll) which will includes the code and can be
loaded within the Tcl execution shell.
*/

#include "tcl.h"
#include "tclDecls.h"
#include <stdio.h>
#include <string.h>

// Initialization Code
#ifdef _WIN32
#define MYFUNCTION_EXPORT __declspec( dllexport )
#else
#define MYFUNCTION_EXPORT
#endif

int MyFunction (ClientData, Tcl_Interp* interp,int objc, Tcl_Obj *CONST objv[])
{
// Code should be placed here
  return TCL_OK;
}
/*
  C++ to Tcl Interface
*/

extern "C" {
int  MYFUNCTION_EXPORT Myfunction_Init(Tcl_Interp* interp);
int  MYFUNCTION_EXPORT Myfunction_SafeInit(Tcl_Interp* interp);
}

int Myfunction_Init(Tcl_Interp* interp)
{
  Tcl_CreateObjCommand(interp, " MyFunction", MyFunction, (ClientData) NULL,
(Tcl_CmdDeleteProc *) NULL);
  return TCL_OK;
}

int Myfunction_SafeInit(Tcl_Interp* interp)
{
  return Myfunction_Init(interp);
}
```

Appendix V: Tcl Code for Different Auxiliary Functions

```
# File Name: AuxiliaryFunctions.Tcl
# Objective: This file contains the customized Tcl procedures that can be used to help
#            accomplish the various tasks needed for the experiments of this thesis.


# Procedure Name: ConvertFromTetgenToVTK
# Objective    : This procedure is used to convert the TetGen output files .node and .ele
#                to .vtk file that can be read back by the VTK Component
# input        : String path to where the generated tetgen .node and .ele are
# output       : Generates .vtk
# example :  ConvertFromTetgenToVTK "C:\\Thesis\\workspace\\Data\\Frog\\spleen.1"

proc ConvertFromTetgenToVTK { strTetGenFile } {

        #  Read TetGen .node file
                set fp [open $strTetGenFile.node r]
                set TetGenNodeData [read $fp]
                close $fp


        # Create a grid object to store the information of the tetgen files
        vtkUnstructuredGrid tetgenGrid

        # Store the points generated by tetgen into a vtkPoints object
        vtkPoints tetgenPoints

                set AllLinesData [split $TetGenNodeData "\n"]
                set bFirstLine "true"
                foreach OneLineData $AllLinesData {
                    set AllWordsData [split $OneLineData " "]
                    set nWordNumber 0
                    foreach OneWordData $AllWordsData {
                        string trim $OneWordData
                        # Check if the line has a comment
                        if {[string index $OneWordData 0] == "#"} {
                                break
                        } else {
                                if {[string length $OneWordData] != 0} {
                                        set arg($nWordNumber) $OneWordData
                                        incr nWordNumber
                                }
                        }
                    }

                    if {$nWordNumber == 4} {

                            if {$bFirstLine == "true"} {
```

```
                    tetgenPoints SetNumberOfPoints $arg(0)
                } else {
                    tetgenPoints InsertPoint $arg(0) $arg(1) $arg(2) $arg(3)
                    }
            }
        set bFirstLine "false"
        }

# Store the points into the grid
tetgenGrid SetPoints tetgenPoints


# Read TetGen .ele file
        set fp [open $strTetGenFile.ele r]
        set TetGenEleData [read $fp]
        close $fp


set testcount 0
# Store the tetrahedra generated by tetgen into a vtkTetra object
# and then store them in the grid

        set AllLinesData [split $TetGenEleData "\n"]
        set bFirstLine "true"


        foreach OneLineData $AllLinesData {
            set AllWordsData [split $OneLineData " "]
            set nWordNumber 0
            foreach OneWordData $AllWordsData {
                string trim $OneWordData
                # Check if the line has a comment
                if {[string index $OneWordData 0] == "#"} {
                        break
                } else {
                        if {[string length $OneWordData] != 0} {
                                set arg($nWordNumber) $OneWordData
                                incr nWordNumber
                        }
                }
            }

            if {$bFirstLine == "true"} {
                    if {$nWordNumber == 3} {
                        # Allocate memory in the grid to store the tetrahedron
                        tetgenGrid Allocate $arg(0) 100
                    }

            } else {
                    if {$nWordNumber == 5} {
                                vtkTetra tetgenTetra
```

```
                                        [tetgenTetra GetPointIds] SetId 0 $arg(1)
                                        [tetgenTetra GetPointIds] SetId 1 $arg(2)
                                        [tetgenTetra GetPointIds] SetId 2 $arg(3)
                                        [tetgenTetra GetPointIds] SetId 3 $arg(4)
                                        tetgenGrid InsertNextCell [tetgenTetra
GetCellType] [tetgenTetra GetPointIds]
                                        tetgenTetra Delete
                                }
                        }

                incr testcount

                set bFirstLine "false"
                }


        vtkUnstructuredGridWriter usGridWriter
          usGridWriter SetInput tetgenGrid
          eval usGridWriter SetFileName $strTetGenFile.vtk
          usGridWriter SetFileType 1
          usGridWriter Update
 }


# Procedure Name: DisplayMesh
# Objective     : This procedure is used to display the mesh of type vtkUnstructuredGrid
#                 : on the screen
# input      : An vtkUnstructuredGrid object
# output     : Display the vtkUnstructuredGrid object on the screen.
# example: DisplayMesh myMesh

proc DisplayMesh { tetgenGrid } {

vtkDataSetMapper aTetraMapper
  aTetraMapper SetInput tetgenGrid

vtkActor aTetraActor
  aTetraActor SetMapper aTetraMapper
  aTetraActor AddPosition 4 0 0
  [aTetraActor GetProperty] SetDiffuseColor 0 1 0

# Create the usual rendering stuff.
vtkRenderer ren1
vtkRenderWindow renWin
  renWin AddRenderer ren1
  renWin SetSize 300 150
vtkRenderWindowInteractor iren
  iren SetRenderWindow renWin

ren1 SetBackground .1 .2 .4
```

```
ren1 AddActor aTetraActor

ren1 ResetCamera
[ren1 GetActiveCamera] Azimuth 30
[ren1 GetActiveCamera] Elevation 20
[ren1 GetActiveCamera] Dolly 2.8
ren1 ResetCameraClippingRange

renWin Render

# render the image
#
iren AddObserver UserEvent {wm deiconify .vtkInteract}
iren Initialize
wm withdraw .
}

# Procedure Name: LoadFrogDataInfo
# Objective    : This procedure is used to set the values of the various parameters
#                needed in the experiment based on the name of the frog tissue
#
# input    : The name of the tissue
# output   : Set the parameters to the data the specified tissue.
# example: DisplayMesh myMesh

proc LoadFrogDataInfo { strTissueName } {

global  ROWS COLUMNS STUDY PIXEL_SIZE SPACING VALUE
global  TISSUE START_SLICE END_SLICE VOI


set ROWS 470
set COLUMNS 500
set STUDY "C:/Feras/Thesis/workspace/Data/Frog/Data/frog/frogTissue"
set PIXEL_SIZE 1
set SPACING 1.5
set VALUE 127.5

switch $strTissueName {

        blood { set TISSUE 1
                set START_SLICE 14
                set END_SLICE 131
                set VOI "33 406 62 425 $START_SLICE $END_SLICE" }

        brain { set TISSUE 2
                set START_SLICE 1
                set END_SLICE 33
                set VOI "349 436 211 252 $START_SLICE $END_SLICE" }

        duodenum { set TISSUE 3
```

```
            set START_SLICE 35
            set END_SLICE 105
            set VOI "189 248 191 284 $START_SLICE $END_SLICE" }

eye_retna { set TISSUE 4
            set START_SLICE 1
            set END_SLICE 41
            set VOI "382 438 180 285 $START_SLICE $END_SLICE" }

eye_white { set TISSUE 5
            set START_SLICE 1
            set END_SLICE 37
            set VOI "389 433 183 282 $START_SLICE $END_SLICE" }

heart { set TISSUE 6
          set START_SLICE 49
          set END_SLICE 93
          set VOI "217 299 186 266 $START_SLICE $END_SLICE" }

ileum { set TISSUE 7
          set START_SLICE 25
          set END_SLICE 93
          set VOI "172 243 201 290 $START_SLICE $END_SLICE" }

kidney { set TISSUE 8
           set START_SLICE 24
           set END_SLICE 78
           set VOI "116 238 193 263 $START_SLICE $END_SLICE" }

l_intestine { set TISSUE 9
              set START_SLICE 56
              set END_SLICE 106
              set VOI "115 224 209 284 $START_SLICE $END_SLICE" }

liver { set TISSUE 10
          set START_SLICE 25
          set END_SLICE 126
          set VOI "167 297 154 304 $START_SLICE $END_SLICE" }

lung { set TISSUE 11
       set START_SLICE 24
       set END_SLICE 59
       set VOI "222 324 157 291 $START_SLICE $END_SLICE" }

nerve { TISSUE 12
          set START_SLICE 7
          set END_SLICE 113
          set VOI "79 403 63 394 $START_SLICE $END_SLICE" }

skeleton { set TISSUE 13
             set VALUE 64.5
```

```
                set START_SLICE 1
                set END_SLICE 136
                set VOI "23 479 8 469 $START_SLICE $END_SLICE"
                set GAUSSIAN_STANDARD_DEVIATION "1.5 1.5 1" }

        spleen { set TISSUE 14
                set START_SLICE 45
                set END_SLICE 68
                set VOI "166 219 195 231 $START_SLICE $END_SLICE" }

        stomach { set TISSUE 15
                set START_SLICE 26
                set END_SLICE 119
                set VOI "143 365 158 297 $START_SLICE $END_SLICE" }

}
}

# The rest of the code is some functions to print out helpful messages
proc readerStart {} {global NAME; puts -nonewline "$NAME read took:\t"; flush stdout};
proc mcubesStart {} {global NAME; puts -nonewline "$NAME mcubes generated\t"; flush
stdout};
proc mcubesEnd {} {
   global NAME
   puts -nonewline "[[mcubes GetOutput] GetNumberOfPolys]"
   puts -nonewline " polygons in "
   flush stdout
};
proc decimatorStart {} {global NAME; puts -nonewline "$NAME decimator generated\t";
flush stdout};
proc decimatorEnd {} {
   global NAME
   puts -nonewline "[[decimator GetOutput] GetNumberOfPolys]"
   puts -nonewline " polygons in "
   flush stdout
};
proc smootherStart {} {global NAME; puts -nonewline "$NAME smoother took:\t"; flush
stdout};

proc writerStart {} {global NAME; puts -nonewline "$NAME writer took:\t"; flush stdout};
```

Appendix VI: Tcl Code to Create Reports About Mesh Quality

```
# File Name: MeshQualityReport.Tcl
# Objective: This file contains the functions needed to report some of the qualities of the
#  mesh.

proc QualityStatics { bReciprocal  IdFile argStaticArray } {

        if {$bReciprocal > 0} {
        puts -nonewline $IdFile  [ expr 1 / [$argStaticArray GetComponent 0 0]]
        } else {
        puts -nonewline $IdFile [$argStaticArray GetComponent 0 0]
        }
        puts -nonewline $IdFile "\t"

        if {$bReciprocal > 0} {
        puts -nonewline $IdFile [ expr 1 / [$argStaticArray GetComponent 0 1]]
        } else {
        puts -nonewline $IdFile [$argStaticArray GetComponent 0 1]
        }
        puts -nonewline $IdFile "\t"

        if {$bReciprocal > 0} {
        puts -nonewline $IdFile [ expr 1 / [$argStaticArray GetComponent 0 2]]
        } else {
        puts -nonewline $IdFile [$argStaticArray GetComponent 0 2]
        }
        puts -nonewline $IdFile "\t"

        #Print to the screean
        puts  ""
        puts  "Min\tAverage\tMax"
        puts -nonewline [$argStaticArray GetComponent 0 0]
        puts -nonewline "\t"
        puts -nonewline [$argStaticArray GetComponent 0 1]
        puts -nonewline "\t"
        puts -nonewline [$argStaticArray GetComponent 0 2]
        puts -nonewline "\t"
        puts  ""

}


proc printline { } {
set NumberOfSeperator 30
        for {set i 0 } { $i < $NumberOfSeperator} {incr i} {
        puts -nonewline "="
        }
        puts ""
}
```

```
proc ReportQualityStatics {strVTKFilePath timeInfo} {

vtkUnstructuredGridReader gridReader
        gridReader SetFileName $strVTKFilePath
        gridReader Update

vtkMeshQuality objMeshQuality
        objMeshQuality  SetInput [gridReader GetOutput]


# The following commented line can be used to print a header to the file
#puts "Tetrahedral quality of mesh: $strVTKFilePath"
#printline
#puts   "Tissue\t\t\t\tAngle\t\t\t\t\t\tAlpha\t\t\t\t\t\tBeta\t\t\t\t\t\tBetaBaker\t\t"
#puts
"Name\t\tMin\t\tAverage\t\tMax\t\tMin\t\tAverage\t\tMax\t\tMin\t\tAverage\t\tMax\t\tMin\t\tA
verage\t\tMax"

        set filename "results1.txt"
        set fileId [open $filename "a+"]

        puts  -nonewline $fileId "$strVTKFilePath\t"
        puts  -nonewline  "$strVTKFilePath\t"

        objMeshQuality          SetTetQualityMeasureToMinAngle
        objMeshQuality  Update
        puts  -nonewline  "\nMinAngle"
        QualityStatics  0 $fileId [[[objMeshQuality GetOutput] GetFieldData] GetArray
"Mesh Tetrahedron Quality"]

        objMeshQuality          SetTetQualityMeasureToAspectBeta
        objMeshQuality  Update
        puts  -nonewline  "\nAspectBeta"
        QualityStatics  1 $fileId [[[objMeshQuality GetOutput] GetFieldData] GetArray
"Mesh Tetrahedron Quality"]

        objMeshQuality          SetTetQualityMeasureToAspectRatio
        objMeshQuality  Update
        puts  -nonewline  "\nAspectRatio"
        QualityStatics  1 $fileId [[[objMeshQuality GetOutput] GetFieldData] GetArray
"Mesh Tetrahedron Quality"]

        puts $fileId ""

        close $fileId

        set filename2 "results2.txt"
        set fileId2 [open $filename2 "a+"]

        puts  -nonewline $fileId2 "$strVTKFilePath\t"
```

```
        puts -nonewline $fileId2 [[[[objMeshQuality GetOutput] GetFieldData] GetArray
"Mesh Tetrahedron Quality"] GetComponent 0 4]
        puts -nonewline $fileId2 "\t"


        puts  -nonewline "Number of Tetrahedrons="
        puts  [[[[objMeshQuality GetOutput] GetFieldData] GetArray "Mesh Tetrahedron
Quality"] GetComponent 0 4]

        objMeshQuality        SetTetQualityMeasureToVolume
        objMeshQuality  Update
        puts  -nonewline  "\nVolume"
        QualityStatics  0 $fileId2 [[[objMeshQuality GetOutput] GetFieldData] GetArray
"Mesh Tetrahedron Quality"]

        puts -nonewline $timeInfo

        puts -nonewline $fileId2 $timeInfo
        puts -nonewline $fileId2 "\t\n"

        close $fileId2

}
```

Appendix VII: Example of a Generated Mesh Quality Report

```
spleen read took:       0.480556 seconds
spleen mcubes generated 5104 polygons in 0.006696 seconds
spleen decimator generated      2040 polygons in 0.069501 seconds
spleen smoother took:  0.013781 seconds
spleen writer took:     0.022524 seconds
Opening spleen.ply.
Constructing Delaunay tetrahedralization.
  Creating initial tetrahedralization.
  Incrementally inserting points.
  15846 Flips (T23 9297, T32 6547, T22 0, T44 2)
Delaunay seconds:  0.078
Creating surface mesh.
  Constructing mapping from indices to points.
  Constructing mapping from points to tetrahedra.
  Unifying segments.
  Constructing mapping from points to subfaces.
  Merging coplanar facets.
  Marking acute vertices.
  Constructing mapping from points to segments.
  1022 acute vertices.
Perturbing vertices.
  0 break points.
Delaunizing segments.
  Constructing mapping from points to tetrahedra.
  Queuing missing segments.
  779 protect points.
  R1: 521,  R2: 206,  R3: 52.
Constraining facets.
  Constructing mapping from points to tetrahedra.
  The biggest cavity: 8 faces, 6 vertices
  Enlarged 0 times
Segment and facet seconds:  0.156
Removing unwanted tetrahedra.
  Marking concavities for elimination.
  Marking neighbors of marked tetrahedra.
  Deleting marked tetrahedra.
Hole seconds:  0
Repairing mesh.
Repair seconds:  0.062
Adding Steiner points to enforce quality.
  Marking sharp segments.
  5484 sharp segments.
  Deciding feature-point sizes.
  Constructing mapping from points to segments.
  1028 feature points.
  779 Steiner feature points.
  Splitting encroached subsegments.
  525 split points.
  Splitting encroached subfaces.
```

```
  15 split points.
  Splitting bad tetrahedra.
  957 refinement points.
  Totally added 1497 points.
Quality seconds:  0.532
Optimizing mesh.
  320 edges are flipped.
  3 passes.
  6 points are inserted (3 on segment).
  1 faces are flipped.
Optimize seconds:  0.062

Writing spleen.1.node.
Writing spleen.1.ele.
Writing spleen.1.face.
Writing spleen.1.smesh.

Output seconds:  0.141
Total running seconds:  1.031

Statistics:

  Input points: 1028
  Input facets: 2040
  Input segments: 3060
  Input holes: 0
  Input regions: 0

  Mesh points: 3310
  Mesh tetrahedra: 13012
  Mesh triangles: 28425
  Mesh subfaces: 4802
  Mesh subsegments: 4438

Mesh quality statistics:

  Smallest volume:      0.00018011  | Largest volume:       13.132
  Shortest edge:        0.052171    | Longest edge:         5.8542
  Smallest facangle:      1.031     | Largest facangle:     169.5270
  Smallest dihedral:    0.89174     | Largest dihedral:     173.3423

  Aspect ratio histogram:
       < 1.5      :    132    |    6 - 10      :    777
     1.5 - 2      :   2100    |   10 - 15      :    199
      2 - 2.5     :   3082    |   15 - 25      :     78
     2.5 - 3      :   2488    |   25 - 50      :     17
      3 - 4       :   2557    |   50 - 100     :      4
      4 - 6       :   1575    |   100 -        :      3
  (A tetrahedron's aspect ratio is its longest edge length divided by its
    smallest side height)
```

Face angle histogram:

0 - 10 degrees:	333	90 - 100 degrees:	4196
10 - 20 degrees:	1461	100 - 110 degrees:	2162
20 - 30 degrees:	4306	110 - 120 degrees:	1052
30 - 40 degrees:	8388	120 - 130 degrees:	383
40 - 50 degrees:	9292	130 - 140 degrees:	123
50 - 60 degrees:	4645	140 - 150 degrees:	53
60 - 70 degrees:	4897	150 - 160 degrees:	10
70 - 80 degrees:	8550	160 - 170 degrees:	7
80 - 90 degrees:	6992	170 - 180 degrees:	0

Minimum input face angle is 1.60076 (degree).

Dihedral angle histogram:

0 - 5 degrees:	58	80 - 110 degrees:	7232
5 - 10 degrees:	290	110 - 120 degrees:	1924
10 - 20 degrees:	1500	120 - 130 degrees:	1425
20 - 30 degrees:	2524	130 - 140 degrees:	1057
30 - 40 degrees:	3298	140 - 150 degrees:	606
40 - 50 degrees:	3204	150 - 160 degrees:	453
50 - 60 degrees:	1882	160 - 170 degrees:	141
60 - 70 degrees:	256	170 - 175 degrees:	1
70 - 80 degrees:	173	175 - 180 degrees:	0

Minimum input facet dihedral angle is 68.8558 (degree).

Memory allocation statistics:

Maximum number of vertices: 3310
Maximum number of tetrahedra: 13294
Maximum number of subfaces: 4802
Maximum number of segments: 6120
Approximate heap memory used by the mesh (K bytes): 1505.33

spleen.1.vtk
MinAngle
Min Average Max
0.888275 42.9754 88.8655

AspectBeta
Min Average Max
1.00455 1.91216 66.5379

AspectRatio
Min Average Max
1.04813 2.12916 62.1592
Number of Tetrahedrons=13012

Volume
Min Average Max
0.00017955 0.442654 13.1318
1.132368

Appendix VIII: Example of How to Execute the Different Components

```
# File Name: experiment.tcl
# Objective: This file is where we connect all the various low-level tasks together
#            to produce the FEM mesh.
#
# Note    :This file is provided as an example to show how the experiment can be done.
#          The actual experimentations presented in this thesis are actually a modified
#          versions of this example that were done by modifying the parameters used in
#          the different procedures(Algorithms), changing the order of some procedure
#          calls, or by replacing the procedures with equivalent ones that implement
#          different algorithms.

lappend auto_path [ file dirname [ info script ]]

if { $tcl_platform(platform) == "windows" } {
   load TclTetGen.dll
} else {
   load libTclTetGen.so
}

#Specify the name of the tissue to do the experiment
if {$argc == 0} {
   puts "Zeroooooooo"
} else {
  puts "More than 0"
  set NAME [lindex $argv 0]
}

source AuxiliaryFunctions.tcl
source MeshQualityReport.tcl

# Set Default values for the different parameters
set PIXEL_SIZE 1
set START_SLICE 1

set FEATURE_ANGLE 60

#Remove Islands Parameters
set ISLAND_ENABLE_FLAG -1
set ISLAND_AREA 4
set ISLAND_REPLACE -1

#Shrinker Parameters
set SHRINKER_ENABLE_FLAG -1
set SAMPLE_RATE "3 3 3"

# Marching Cubes parameter
set MC_ENABLE_FLAG 1
```

```
# Decimate parameters
set DECIMATE_ENABLE_FLAG 1
set DECIMATE_ANGLE $FEATURE_ANGLE
set DECIMATE_REDUCTION .6

# Smooth parameters
set SMOOTH_ENABLE_FLAG 1
set SMOOTH_ANGLE $FEATURE_ANGLE
set SMOOTH_ITERATIONS 10
set SMOOTH_FACTOR .01

# GAUSSIAN smooth parameters
set GAUSSIAN_ENABLE_FLAG -1
set GAUSSIAN_STANDARD_DEVIATION "2 2 2"
set GAUSSIAN_RADIUS_FACTORS "1 1 1"

# TetGen parameters
set TETGEN_ENABLE_FLAG 1
set TETGEN_SWITCH pq1.414V

#Load Specific Tissue parameters
LoadFrogDataInfo $NAME

# Coordinate Computations
set originx [expr ( $COLUMNS / 2.0 ) * $PIXEL_SIZE * -1.0]
set originy [expr ( $ROWS / 2.0 ) * $PIXEL_SIZE * -1.0]
set minx [lindex $VOI 0]
set maxx [lindex $VOI 1]
set miny [lindex $VOI 2]
set maxy [lindex $VOI 3]
set minz [lindex $VOI 4]
set maxz [lindex $VOI 5]

# adjust y bounds for PNM coordinate system
set tmp $miny
set miny [expr $ROWS - $maxy -1]
set maxy [expr $ROWS - $tmp -1]

# reader reads slices
vtkPNMReader reader;
   reader SetFilePrefix $STUDY
   reader SetDataSpacing $PIXEL_SIZE $PIXEL_SIZE $SPACING
   reader SetDataOrigin $originx $originy [expr $START_SLICE * $SPACING]
   eval reader SetDataVOI $minx $maxx $miny $maxy $minz $maxz
   [reader GetOutput] ReleaseDataFlagOn
   set lastConnection reader

if {$ISLAND_ENABLE_FLAG >= 0} {
vtkImageIslandRemoval2D islandRemover
    islandRemover SetAreaThreshold $ISLAND_AREA
    islandRemover SetIslandValue $ISLAND_REPLACE
```

```
    islandRemover SetReplaceValue $TISSUE
    islandRemover SetInput [$lastConnection GetOutput]
    set lastConnection islandRemover
}

vtkImageThreshold selectTissue
  selectTissue ThresholdBetween $TISSUE $TISSUE
  selectTissue SetInValue 255
  selectTissue SetOutValue 0
  selectTissue SetInput [$lastConnection GetOutput]
  set lastConnection selectTissue

if {$SHRINKER_ENABLE_FLAG >= 0} {
vtkImageShrink3D shrinker
    shrinker SetInput [$lastConnection GetOutput]
    eval shrinker SetShrinkFactors $SAMPLE_RATE
    shrinker AveragingOn
    set lastConnection shrinker
 }

if {$GAUSSIAN_ENABLE_FLAG >= 0} {
  vtkImageGaussianSmooth gaussian
     eval gaussian SetStandardDeviations $GAUSSIAN_STANDARD_DEVIATION
     eval gaussian SetRadiusFactors $GAUSSIAN_RADIUS_FACTORS
     gaussian SetInput [$lastConnection GetOutput]
     set lastConnection gaussian
}


if {$MC_ENABLE_FLAG >= 0} {
#vtkMarchingCubes mcubes;
vtkContourFilter mcubes;
   mcubes SetInput [$lastConnection GetOutput]
   mcubes ComputeScalarsOff
   mcubes ComputeGradientsOff
   mcubes ComputeNormalsOff
   eval mcubes SetValue 0 $VALUE
   [mcubes GetOutput] ReleaseDataFlagOn
   set lastConnection mcubes
}

if {$DECIMATE_ENABLE_FLAG >= 0} {

vtkDecimatePro decimator
   decimator SetInput  [$lastConnection GetOutput]
     decimator SetTargetReduction $DECIMATE_REDUCTION
     decimator PreserveTopologyOn
     [decimator GetOutput] ReleaseDataFlagOn
     set lastConnection decimator
}
```

```
# The following can be used as an alternate Decimator algorithm
#vtkQuadricDecimation Newdecimator
#  Newdecimator SetInput [$lastConnection GetOutput]
#  Newdecimator SetTargetReduction .95
#  Newdecimator AttributeErrorMetricOn
#  set lastConnection Newdecimator


#vtkCleanPolyData cleaner
#        cleaner SetInput [$lastConnection GetOutput]
#        cleaner SetTolerance 0.005
#     set lastConnection cleaner


if {$SMOOTH_ENABLE_FLAG >= 0} {
vtkSmoothPolyDataFilter smoother
   smoother SetInput [$lastConnection GetOutput]
   eval smoother SetNumberOfIterations $SMOOTH_ITERATIONS
   eval smoother SetRelaxationFactor $SMOOTH_FACTOR
   eval smoother SetFeatureAngle $SMOOTH_ANGLE
   smoother FeatureEdgeSmoothingOff
   smoother BoundarySmoothingOff;
   smoother SetConvergence 0
   [smoother GetOutput] ReleaseDataFlagOn
   set lastConnection smoother
}




# Write Data as .ply file
vtkPLYWriter writer
   writer SetInput [$lastConnection GetOutput]
   eval writer SetFileName $NAME.ply
   writer SetFileType 1
   set lastConnection writer


#Note that we Add observers for the various steps to record time
reader AddObserver StartEvent readerStart
puts "[expr [lindex [time {reader Update;} 1] 0] / 1000000.0] seconds"


if {$MC_ENABLE_FLAG >= 0} {
mcubes AddObserver StartEvent mcubesStart
mcubes AddObserver EndEvent mcubesEnd
puts "[expr [lindex [time {mcubes Update;} 1] 0] / 1000000.0] seconds"
}
```

```
if {$DECIMATE_ENABLE_FLAG >= 0} {
decimator AddObserver StartEvent decimatorStart
decimator AddObserver EndEvent decimatorEnd
puts "[expr [lindex [time {decimator Update;} 1] 0] / 1000000.0] seconds"
}

if {$SMOOTH_ENABLE_FLAG >= 0} {
smoother AddObserver StartEvent smootherStart
puts "[expr [lindex [time {smoother Update;} 1] 0] / 1000000.0] seconds"
}

writerStart
puts "[expr [lindex [time {writer Update;} 1] 0] / 1000000.0] seconds"

if {$TETGEN_ENABLE_FLAG >= 0} {

# Start TetGen

set TetGenTime [expr [lindex [time {TetGen $TETGEN_SWITCH  $NAME.ply;} 1] 0] /
1000000.0]

# Convert the resulted .node and .ele to .vtk
ConvertFromTetgenToVTK $NAME.1

# WriteUp Reports
ReportQualityStatics $NAME.1.vtk $TetGenTime

}

exit
```