Imperial College of Science,
Technology and Medicine
Department of Computing

Building of three-dimensional vascular models for interactive simulation in Interventional Radiology

by

Daniel Ryan King

Submitted in partial fulfilment of the requirements for the MSc Degree in Advanced Computing of Imperial College London

Abstract

Title: Building of three-dimensional vascular models for interactive simulation in Interventional Radiology

Author: Daniel Ryan King

Degree: MSc in Advanced Computing, Imperial College London

Supervisors: Dr. Fernando Bello and Dr. Vincent Luboz

Objectives: To provide a set of tools for generating 3D vascular models from patient datasets for interventional radiology simulations.

Background: As interventional radiology procedures become more common place, improved training methods must be developed to better prepare surgeons. Virtual reality simulators provide a safe and effective way for surgeons to gain the required experience. These simulations use models of the involved anatomy to provide visualisation and interaction. A set of tools to generate these 3D anatomical models from patient datasets will improve the variety of situations available to trainees, as well as allow experienced surgeons to practise specific procedures.

Methods: An application was developed in C++ to provide access to a set of interactive modelling tools using supporting functionality from FLTK, VTK, ITK, CGAL, and TetGen. Several segmentation algorithms were integrated and extended to allow the extraction of structures from datasets. A 2.5D extension of Lewis Griffin's hierarchical segmentation was developed. Another segmentation method using the Hessian matrix was also adapted to provide a fast method for finding vessel-like structures in an image. A distance transform skeletonisation algorithm was included to create a compact set of skeletal points from a segmented vessel. These skeletal points can then be connected together to form an approximate centreline via a minimum spanning tree algorithm. Mesh generation algorithms allow the creation of triangular and tetrahedral meshes from the segmented datasets.

Results: Experiments were conducted to test the effectiveness of the included algorithms. Hierarchical segmentation was found to be a good alternative to level set methods for vessels and other high contrast structures. By combining a vesselness measure, calculated from the Hessian matrix, with a fast marching segmentation, the vessels of a dataset can be quickly and automatically extracted. This method was able to segment even the many small vessels around the aortic arch. The results from skeletonisation were validated as good representations of the original vessels by comparing the reconstruction with the actual vascular structure. Different meshing algorithms, such as decimation and smoothing, were found to cause minimal deviation from an accurate representation of the anatomical structures.

Conclusions: The structure of the program allows the user to easily select and combine tools in a sequence to generate anatomical models from specific patient datasets. The tools also allow the user to interactively improve the results from segmentation and mesh generation so that the models meet the simulation's needs and requirements. Future algorithms can be easily added to provide improved and extended functionality.

Acknowledgements

First, I would like to thank Dr. Fernando Bello for his guidance during this exciting and important research project. His suggestions and feedback were an integral part to expanding and developing the project.

I thank Dr. Vincent Luboz for motivating me with weekly goals and overseeing the many complex algorithms. He was always happy to help whenever problems were encountered. His experience in vascular segmentation and skeletonisation was also invaluable.

Dr. Pierre-Frederic Villard provided non-vascular applications of the project's segmentation methods and provided insight into different mesh generation algorithms. His work in liver and breathing simulations also further tested the program, allowing improvements to be made.

Sincere appreciation goes to Nizar Din and Laura Tucker for providing ideas and feedback for the project's user interface. Their involvement in data collection and testing proved essential for improving hierarchical segmentation.

Special thanks to Dr. Eddie Edwards and all the other researchers and students working at Saint Mary's Hospital for understanding the problems and listening to the ideas presented at the bi-weekly meetings. They asked insightful questions and gave helpful comments.

Last, but not least, I thank my family for their unconditional support and encouragement.

Table of Contents

1 - Introduction

1.1 - Interventional Radiology

Interventional radiology is a form of minimally invasive surgery in which images are used to plan and guide the procedures [1]. These images can come from a variety of sources including X-ray radiographs, computed tomography (CT), and magnetic resonance imaging (MRI) [2]. Interventional radiologists use specialized tube- or rod-like instruments that are inserted into the body through small incisions or natural openings. Cardiovascular procedures are commonly performed with these methods using the structure of the vascular system to access the target area. A catheter can be inserted through a small incision in the groin into the femoral artery and then manoeuvred to the desired location [3]. Because it is less invasive than traditional open surgery, interventional radiology has many advantages including less bleeding, less pain, less need for anaesthesia, and shorter recovery time [3][4].

However, interventional radiological procedures are complex and pose several difficulties. The minimal invasiveness of these procedures forces all manipulation of the instruments to take place outside of the body with restricted movement and poor force-feedback [4]. The interventional radiologist also has no direct sight to the target area, so images of the internal organs, vasculature, and instruments must be displayed on video monitors using imaging methods such as X-ray fluoroscopy or ultrasound [3]. Training is very important to ensure interventional radiologists develop the required hand-eye coordination to safely control the instruments while watching the movement on a display. The procedures are composed of many steps and many points at which a decision must be made [5]; trainees must learn to make the correct decisions, especially when complications arise. Experts also need methods of rehearsing complicated procedures before entering the operating room.

Training methods based on physical models do not provide the realism or variety of anatomy needed to fully prepare interventional radiologists [4], and training on animals is costly and raises ethical issues [5]. The traditional surgical apprenticeships require a long time for interventional radiologists to become fully prepared and put patients at potential risk during training [3]. Ideally, most training should take place outside of the operating room where procedures can be repeated and mistakes corrected. These training methods also lack objective assessment of a trainee's performance which would allow more focus on the areas that need improvement [5]. Virtual reality training simulations have been proposed to help solve these problems.

1.2 - Virtual Reality Simulation

Virtual reality simulation is used to create a computer-generated environment in which the user can realistically interact with objects [3]. These simulators have proved useful when applied to training [6], and they allow trainees to practise potentially dangerous procedures without risking harm to any patient. Simulators can also provide objective feedback on the trainee's improvement, whereas physical models are typically limited to subjective assessments of performance [5]. Simulation also has the benefit of permitting the level of difficulty to be tailored to a specific trainee [3]. There are no time constraints on training and procedures can be repeated as necessary, whereas mistakes are not allowed and time is limited for training on animals or patients. Furthermore, these advantages also often make simulators a cheaper alternative to traditional training methods.

For interventional radiology, simulators can provide a training environment that does not put patients at risk, yet can still allow interventions to be practised under realistic conditions. Haptic devices are usually used to provide force-feedback to the user, simulating the feel of surgical instruments [7]. Video displays of the virtual environment provide a view into the simulation, and stereoscopic display devices can be used to improve the user's depth perception [8]. Various procedures with different anatomies and complications can be simulated to allow trainees to experience and learn from a larger variety of situations before working with real patients. The use of these simulators is not limited to training, as they are also useful for the planning and rehearsal of procedures using patient-specific datasets. An interventional radiologist can determine potential difficulties and find solutions before

entering the operating room and beginning the procedure on the actual patient.

1.3 - 3D Anatomical Modelling Framework

A 3D anatomical modelling framework was developed to generate models of anatomy from patient datasets. The generated models are suitable for use in virtual reality simulations for the training, planning, and rehearsal of interventional radiological procedures. Modern algorithms were employed and improved to allow the visualization, extraction, and meshing of these structures by building upon the functionality provided by several toolkits and libraries.

The patient data sets are volumetric images, typically generated with computed tomography (CT), magnetic resonance imaging (MRI), or X-ray radiographs [2]. These images contain three dimensional, point-sampled information about a specific region of interest of the patient's anatomy. The meaning of each point or voxel in the volume differs according to the imaging method; for example, the values of a CT scan represent the transparency to X-rays of the material at that point [9]. These values are typically mapped to grey-scale intensities and opacities so that the image can be viewed. Volume rendering can allow a visualization of the entire data set, but cutting 2D slices through the volume is an intuitive way to see the structures within [2]. A series of slices allows the user to easily move through the image and build a mental picture of the internal structures. These slices also allow the user to define features in the image by marking specific points. The process of labelling the structures or regions in a image is called segmentation.

Object extraction or segmentation tools allow specific objects or parts of the anatomy to be identified and separated from the rest of the volume [2]. By segmenting the object, it can be viewed and manipulated separately. These tools may be manual, automatic, or an interactive combination. Manual segmentation is a long process requiring the user to delineate objects one slice at a time; however, the results can be as accurate as desired. Fully automatic segmentation algorithms require less time and effort from the user, but the results may not be as expected. Semi-automatic, interactive segmentation algorithms give the user some control and often provide good results. These algorithms allow the segmentation to be improved with interaction from the user, which reduces the amount of manual editing needed after segmentation. Hierarchical segmentation [10] is an excellent example of such an algorithm that lets the user mark interior and exterior points until satisfied with the results. By building upon this algorithm, an intuitive interactive 3D segmentation algorithm was developed that competes well with level set methods in terms of speed and accuracy.

Some segmentation algorithms are better suited for different types of anatomy. Skeletonisation focuses specifically on extracting tubular shapes such as blood vessels. These structures are needed for interventional radiology simulations, and useful information can also be obtained such as the length and radius of such vessels. A distance transform method [11] was used to construct a compact set of skeletal points, and an algorithm was developed to link these points together to form an approximate centreline of segmented vessels. The centreline algorithm is based largely on minimum spanning tree algorithms, but also performs further processing to remove extraneous points, improving the results. A line filter based on the Hessian operator has also been included to automatically extract vessels of specified radii from an image. Although the results from this filter alone are often not sufficiently accurate, they can serve as input to the other segmentation algorithms, reducing the time and effort required by the user to generate quality segmentations of blood vessels.

Even complex segmentation algorithms rarely provide perfect results, so editing functionality allows users to improve the results by removing undesired objects or artefacts from the patient dataset. These editing tools also allow the resulting segmentations to be improved manually if needed. Once a good quality segmentation is complete, the surface of the object is well defined and a representative mesh can be generated.

Meshing algorithms allow the creation of triangular meshes that represent the surface of segmented objects [2]. These meshes provide a compact representation which is useful for visualization, navigation, and simulation. Unlike the volumetric data, triangular meshes can be efficiently rendered by common graphics hardware. The addition of polygonal editing tools allows the user to improve the resulting mesh or modify features as desired. For use in simulations, an internal mesh structure, composed

of tetrahedra, can be generated to represent the volume of the object [13]. This internal structure allows the simulation of a deformable object that a user can interact with in real time.

The program combines these steps into a single cross-platform application written in the C++ programming language. Several existing toolkits and libraries were used to provide supporting functionality. FLTK (http://www.fltk.org/) allowed easy design and creation of the user interface [14]. ITK (http://www.itk.org/) is used for input and output of the volumetric data sets and for some image filtering and segmentation algorithms [15]. VTK (http://www.vtk.org/) provides visualization of the data through direct volume and polygonal mesh rendering [16]; it also includes a wide variety of useful geometric algorithms. CGAL (http://www.cgal.org/) provides more advanced geometric algorithms, including surface mesh generation [17]. Finally, TetGen (http://tetgen.berlios.de/) constructs the inner mesh structures based on the surface meshes of objects [18]. The building of the application, toolkits, and libraries on different platforms is managed with CMake (http://www.cmake.org/) [19], while a Windows installer was created with NSIS [20] and documentation was generated from the code with Doxygen [21].

2 - Background

2.1 - Minimally Invasive Surgery

Minimally invasive surgery is performed through a few small incisions or through natural body openings, rather than the single large incision used in traditional open surgery [22]. The incisions made for minimally invasive surgery are between only a quarter-inch and a half-inch long and allow access to the organs with special tube- or rod-like instruments (Figure 2.2). All manipulation of the instruments takes place outside the body, reducing the invasiveness [4]. Since surgeons have no direct sight into the body, various tools are used to transmit an image of the inside of the body to high resolution video monitors. Surgeons can insert fibre-optic lights (Figure 2.1) and miniature video cameras or endoscopes into the body to view target areas [22]. Alternatively, external tools such as ultrasound or X-ray fluoroscopy can provide a view of the internal structure and tool positions [3]. Surgeons must rely upon these images to guide their instruments and perform procedures.

Figure 2.1: Endoscope with camera and light source [23]

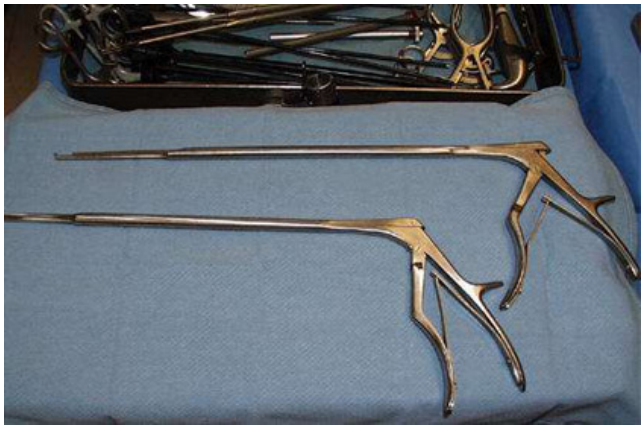


Figure 2.2: Endoscopic instruments [23]

Because of the use of small incisions, minimally invasive surgery has several advantages. Less trauma is caused to the body which leads to less blood loss, smaller surgical scars, and less pain [4]. Typically only local anaesthesia is required, whereas open surgery usually requires general anaesthesia. Patients are able to recover from the surgery quicker, leave the hospital sooner, and return to normal activity earlier [3]. Due to less hospital time, minimally invasive surgery is also often less expensive than traditional open surgery. These benefits have caused minimally invasive surgery to grow in popularity, and surgeons are performing more procedures with minimally invasive techniques.

2.2 - Interventional Radiology

Interventional radiology is an area of minimally invasive procedures that uses elements of radiology. Radiology is the study of images of the human body, and interventional radiologists use these images to plan and guide their procedures [1]. Images of the body may be obtained from several sources, including X-ray radiographs, X-ray fluoroscopy, ultrasound, computed tomography (CT), and magnetic resonance imaging (MRI) [2]. Interventional radiologists use these maps of the internal structures to guide their specialized instruments to the target area and perform the diagnosis or treatment.

Cardiovascular procedures lend themselves well to interventional radiology procedures due to the structure of the vascular system. A small incision in the groin gives access to the vasculature system through the femoral artery [3]. From this entry point, the interventional radiologist can manoeuvre a tube-like tool called a catheter to the desired target area. Fluoroscopic imaging during the intervention provides a visualization of the catheter for navigation. Common procedures include angiography, draining, balloon angioplasty, vascular stenting, coil embolization, and radiofrequency ablation [3]. Angiography is the imaging of blood filled structures injected with a contrast agent to provide a map of the target area for diagnosis or for planning treatment. Catheter tubes can be used for both the injection and draining of fluids from target areas. Balloon angioplasty (Figure 2.4) uses a balloon-tipped catheter to open blocked

blood vessels, after which a wire-mesh tube called a stent may be permanently inserted to keep the blood vessel open. Blood vessels can also be blocked with coil embolization to prevent bleeding or the rupturing of aneurysms. Radiofrequency ablation uses energy to destroy target tissue, which may include abnormal electrical pathways in heart tissue or tumour masses.

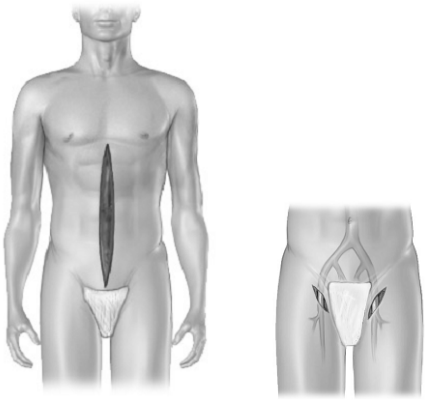


Figure 2.3: Invasiveness of conventional open surgery (left) compared to interventional radiology (right) [3]

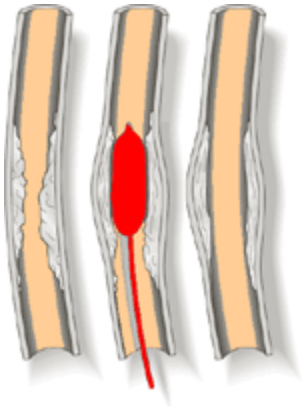


Figure 2.4: Balloon angioplasty procedure [24]

Being a minimally invasive surgery (Figure 2.3), interventional radiology receives the same benefits; however, it also inherits difficulties. The catheter (Figure 2.5) introduces a new set of challenges to interventional radiologists. It is a long flexible tool, which is difficult to manoeuvre through the vascular system [3]. Friction against the vessel wall adds further to this difficulty [4]. Moreover, the catheter gives poor feedback of the forces at the tip which can harm the patient if the interventional radiologist is inexperienced. No direct sight is available, so one must rely solely upon imaging methods which typically provide only a 2D visualization and no depth perception (Figure 2.6).

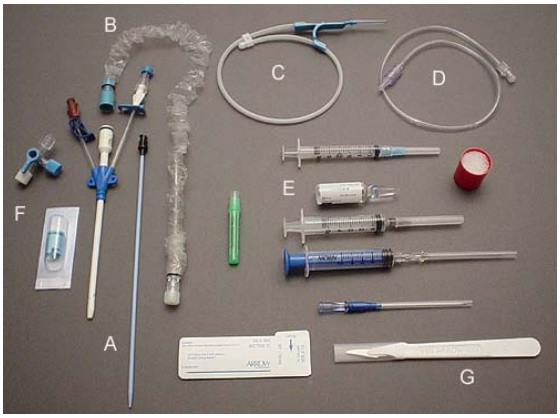


Figure 2.5: Central venous catheter kit [25]

Figure 2.6: X-ray fluoroscopic view of a catheter in the renal artery [26]

All control of the instruments takes place outside the body (Figure 2.7), further disrupting hand-eye coordination [3]. The instruments themselves are long, thin, and unnatural, providing only a limited degree of freedom. Thus, mastering the required skills for these complex procedures is difficult, putting much pressure on training [1].

Figure 2.7: Interventional radiologist using X-ray fluoroscopy to guide a catheter during a procedure [27]

2.3 - Current Training

The main strategy of training for interventional radiologists is "learning by doing" [3]. This is the strategy of the traditional model of surgical apprenticeships, during which supervised training takes place with actual patients. Over time, the trainee gains the necessary skills and will no longer require supervision. However, this method of training can bring the patients at risk due to lack of experience. Furthermore, this process is time-consuming and costly. Alternative training methods have been introduced to keep as much training as possible out of the operating room and at a lower cost.

Lectures, live-case demonstrations using telecommunication, and observation of procedures in the intervention room give trainees experience with more variety of procedures and patients [3]. These methods do not bring risk to patients and can lessen the time and cost of training; however, the trainee does not acquire the first hand experience needed.

Anatomical models and vascular phantoms allow trainees to handle the instruments and practice procedures without actual patients. Phantoms (Figure 2.8 and 2.9) are particularly useful for repeated practise; however, they are restricted to only one example of vasculature geometry [3]. The realism needed to reduce the number of slips while handling actual patients is also lacking [5].

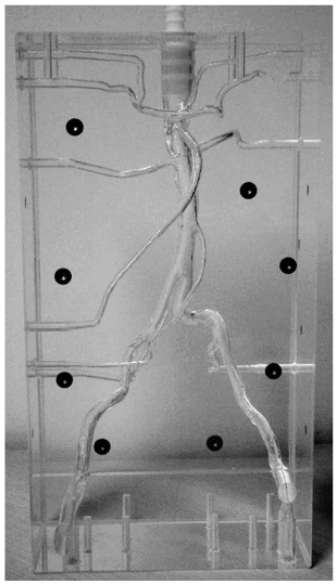


Figure 2.8: An abdominal aorta phantom [3]

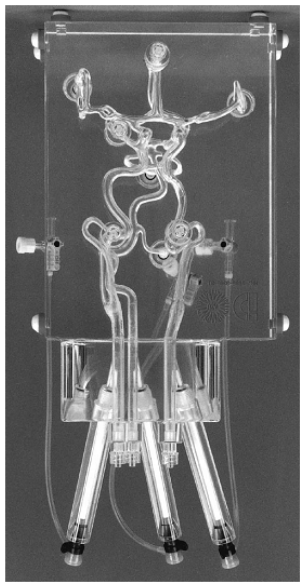


Figure 2.9: A cerebral vasculature phantom [3]

Live animals are currently the best available training method [4], as they provide realistic variations in anatomy, along with realistic interactions with instruments. However, animals are expensive

and do not fully represent human anatomy. The use of live animals in training also raises ethical questions. Cadavers are an anatomically correct substitute; however, the properties and interactions with instruments will not be as accurate because of the rigidity of the cadaver and lack of blood flow. Cadavers are also expensive and limited in supply. The use of live animals and cadavers also exposes trainees to excess radiation during intra-operative imaging that should be avoided [3].

These alternative training methods all have difficulty with objective performance assessment to track a trainee's progress. Performance is key to successful outcomes [5], so tracking improvement is a very important aspect of training. The primary goal of training is to improve the efficiency of a procedure and to reduce the number of errors [4]. These errors can be divided into slips, lapses, and mistakes which correspond to three levels of cognitive behaviours: skill-based, rule-based, and knowledge-based [5]. Each level must be considered during training, and different types of training are better suited for different levels.

The motor skills needed to guide and use the instruments make up the skill-based level [5]. These skills can be effectively trained with low cost skill boxes and phantoms; however, more realistic anatomical simulators are needed to decrease the number of slips related to handling with actual patients [4]. The rule-based level consists of the operation protocol that an interventional radiologist must follow during procedures [5]. These rules can be taught with relatively cheap text books, lectures, and the Internet. Phantoms and animal labs can further solidify these rules if the protocols are enforced during practise. The knowledge-based level is necessary for the interventional radiologist to deal properly with a serious complication during an intervention [5]. However, this knowledge is the most difficult to train as accurate models of individual variations in anatomy and pathology are needed, along with realistic interactions between instruments and a dynamic vascular wall. Unfamiliar random events are required to further prepare trainees for the complications encountered in actual procedures. Fellowship and apprenticeship programs are the traditional methods to meet these requirements, yet at high cost in both money and time [3][4].

Because of the complexity of interventional radiology, many tasks are required for any single procedure. For example, a task as simple as arterial needle puncture requires 101 steps, with 24 decision-making points [5]. Training for every possibility is simply impossible, but trainees should experience a wide enough variety to prepare them for unexpected complications in real patients. Traditional training methods put patients at risk with inexperienced trainees, but alternatives have shortcomings of their own. Virtual reality simulation has been proposed to help overcome these problems.

2.4 - Virtual Reality Simulation

A simulation is an imitation of some real object, state of affairs, or process [3]. Simulations often provide insight into complex systems through experimentation with different scenarios. Computer simulations mathematically model an actual or theoretical system which can be executed to provide output. This output can then be analysed to discover information about the involved processes and interactions. The main advantage of simulations is that they provide insight into possible behaviour and outcomes without significant time and resources. However, simulations have limitations. Because of the complexity of the modelled systems, accurate representation is not always feasible. A high level of detail requires more time for execution, but as details are omitted, the results become less accurate. Furthermore, the data needed to describe the behaviour of the system may be incomplete, leading to more inherent inaccuracy. The trade-off between accuracy and computation time is especially relevant for virtual reality simulations.

Virtual reality is the computer-generated representation of an environment that allows sensory interaction in order to give the impression that the represented environment is real and actually there [4]. Simulation and virtual reality are often combined so that the user can interact with virtual objects, which are simulated realistically. Virtual reality simulators often make use of special purpose hardware, including haptic interfaces to provide tactile and force-feedback to the user [7]. All feedback, including visualization, must be performed in real-time for natural user interaction, imposing a real-time constraint on the underlying simulations [28].

Using virtual reality simulators as a training tool for interventional radiology has several

advantages. First, they allow more training to take place outside of the operating room, rather than putting patients at risk [5]. Thus, scenarios can be repeated as many times as necessary, whenever necessary, and trainees can learn from their mistakes [3]. The variety of anatomical structures is not limited by physical models, animals, or patients, providing more opportunities for trainees to gain experience. Simulators also provide the ability to objectively quantify performance, allowing for easier tracking of skills [5]. The complexity and difficulty of a scenario can be customized to a trainee's current level of experience. [3] The haptic devices of simulators reinforce motor skills, while operation protocol can also be enforced and assessed. Simulators may lack the realism and accuracy to fully prepare trainees, but being better prepared in basic skills will lead to overall improvement [5]. Complex and rarely encountered procedures or scenarios can also be practised, not only by trainees but also by expert interventional radiologists using patient-specific data. Simulators are also cheaper and more ethical than operations on live animals or supervision with actual patients.

Virtual reality simulators have proven effective for training a variety of medical procedures [6]. Simulation has been successfully applied to anaesthesia training and the development of teamwork and crisis management skills [29]. A study has shown that physicians trained to perform retinal photocoagulation by using a virtual reality simulator performed the procedure with similar accuracy compared to physicians trained on actual patient cases [30]. Virtual reality simulators have also been shown to improve the training of lumbar puncture procedures [31] and laparoscopic psychomotor skills for minimally invasive procedures [7]. Virtual reality simulators for the training of interventional radiology procedures can build upon these previous successes.

Several research groups have been developing interventional radiological simulators in recent years. One of the first simulators is the Dawson-Kaufman simulator developed by HT Medical Inc. [32]. Immersion Medical Inc.'s CathSim is currently used to train nursing students to perform venipuncture [33]. CIMIT developed the ICTS training system for interventional cardiology and VIRGIL-a for chest tube insertion training [34]. Several transportable table-top simulators, including those by Xitact Inc. [35], have been built for intravascular procedures. Simulators have also been used specifically for remote procedures by a group of researchers of the Singapore Bioimaging Consortium [1].

Researchers at the Centre for Advanced Studies, Research, and Development in Sardinia have developed a volumetric virtual environment for simulating catheter procedures [8]. They decided to use a volumetric model (Figure 2.11) rather than surface reconstruction because of the characteristics of the needle insertion procedure. A PHANToM device (Figure 2.10) with three degrees-of-freedom is used to provide haptic feedback while manipulating the needle. Their system also uses a head-tracked stereoscopic view so that the projection follows head movements to make the virtual patient appear at a fixed position.

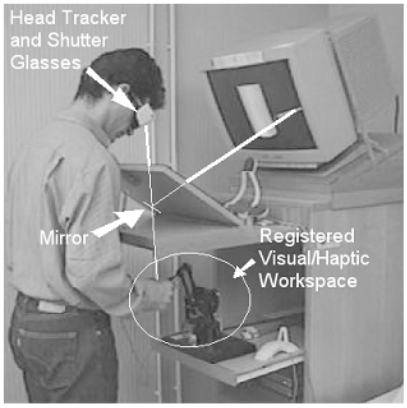


Figure 2.10: Haptic workbench configuration [8]

Figure 2.11: A user looks down into a mirror through shutter glasses to perceive the virtual image [8]

Though simulations can improve basic skills, they can never fully replace real-life patient-based training. The interactions between the doctor and patient must always remain an integral part of advanced training [3]. However, training basic skills in an environment where the patient is not at risk is always preferred, especially at the early stages of a training program. These basic skills also provide strong

foundations for trainees to become more proficient in advanced skills. Simulators can therefore become an important and integral part to the training process. Expert interventional radiologists can also continue to use simulators to practise and prepare for procedures with patient-specific data sets.

2.5 - Medical Imaging

The images of the structures within the body used to plan and guide interventional radiological procedures are often made up of high resolution, volumetric data. This data can come from several sources including computed tomography (CT), magnetic resonance imaging (MRI), rotational angiography, and 3D ultrasound [2]. Medical imaging is the visualization and manipulation of medical data, and several specialized methods exist for handling these volumetric images.

Volumetric images are typically made up of voxels: each voxel represents a point sample in 3D space [9]. The value of a voxel represents a property of the material sampled at that point; for example, the values of voxels resulting from a CT scan represent the opacity of the material to X-rays. These values are usually mapped via a transfer function to optical properties, such as opacity or colour. Voxels form a grid, regular or not, in 3D space, much like pixels form a grid in 2D space. Since most display devices can only display 2D information, the 3D volumetric image must be manipulated to be visualized.

The simplest method to view a volumetric image is to treat it as a series of 2D slices and to view one slice at a time, as shown by Figure 2.12. This method is called multiplanar reformation [2]. These slices can be taken in any direction, parallel to a face of the volume or obliquely. The slices are most often in the axial (transverse) plane, which corresponds to slices taken from head to toe. Slices can also be taken in the sagittal plane, from left shoulder to right shoulder, or the coronal (frontal) plane, from the nose to back of the head.

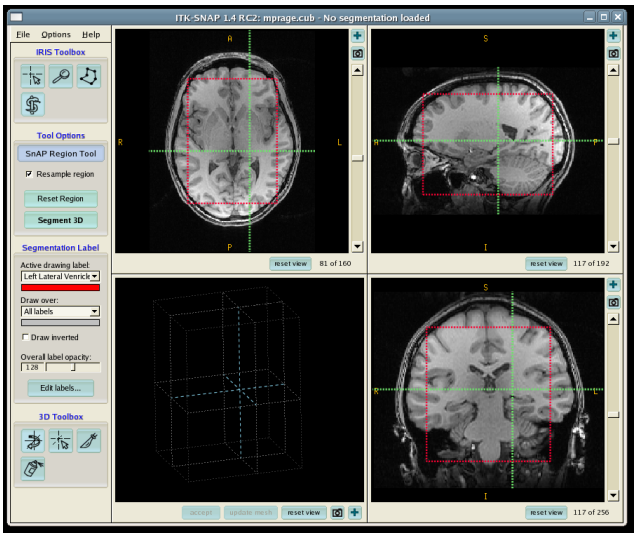


Figure 2.12: A screenshot from ITK SNAP showing the three orthogonal cutting planes [36]

The use of a series of slices is currently the most popular visualization technique, as interventional radiologists can move from slice to slice and build a 3D mental model of the anatomy [2]. Slices also allow the easy selection of points or features in the image. However at times, slices are not practical because of complex or branching structures such as blood vessels which are difficult to follow from slice to slice. In these cases, a 3D view may prove more useful.

Direct volume rendering [9] allows a visualization of the entire data set (Figures 2.13 and 2.14), and there are several rendering methods. Ray casting is a high quality rendering method which shoots a ray into the volume for each pixel in the output image: the reflected light visible at each pixel is calculated from the opacity and colour of a set of samples along the ray [2]. Another high quality method, splatting does the opposite: the voxels are projected onto the image plane to determine the colour of each pixel. Both ray casting and splatting give excellent quality; however, the render times are large due to the computation involved. Lower quality, faster alternatives are provided by shear-warp and texture-based

rendering methods. Shear-warp rendering uses a shear to align the volume so that a line of voxels can project directly to a pixel, and then an image-warp transformation compensates for the shear to produce the final image. Texture-based methods use slices of the volume as textures for polygon meshes which are composited into the final image with graphics hardware. This trade-off between quality and computation should be considered alongside the proposed use of the volume visualization in order to choose the appropriate rendering method.

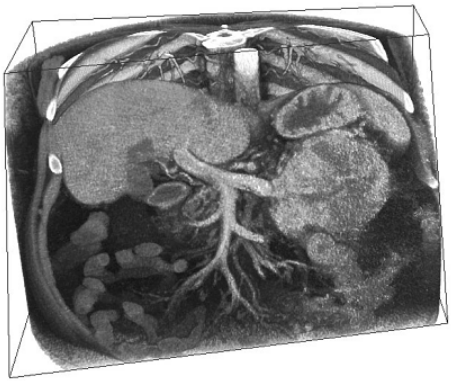


Figure 2.13: Direct volume rendering with a clip plane, allowing the inner anatomy to be seen [9]

Figure 2.14: Direct volume rendering of segmented kidney [9]

Visualization of the entire data set can be useful, but the different regions or parts of the anatomy in the volumetric image need to be identified so that they can be treated individually. This identification is done through the segmentation of the objects [2]. Manipulation and simulation of anatomical structures is also difficult for volumetric images because of the large amount of data involved. For real-time simulation, triangular meshes of the desired objects are needed for efficient rendering by graphics hardware and for efficiently simulating deformation [37].

Because of the recent prevalence of medical imaging devices, several systems have been developed for the visualization and manipulation of this data. Commercial products include systems from Siemens (http://www.medical.siemens.com/) and GE (http://www.gehealthcare.com/), CoActive EXAM-PACS (http://www.coactiv.com/), Viatronix V3D (http://www.viatronix.com/), Materialise SurgiCase (http://www.materialise.com/), Median CT (http://www.mediantechnologies.com/), and 3D-DOCTOR (http://www.ablesw.com/). These commercial systems find much use in hospitals around the world, but researchers have worked to improve upon them with new systems. The Harvard AstroMed Project (http://astromed.iic.harvard.edu/) provides a unique combination of medical (Figure 2.15) and astronomical imaging into a single package [38]. Other projects have focused on medical imaging or just a specific functionality.

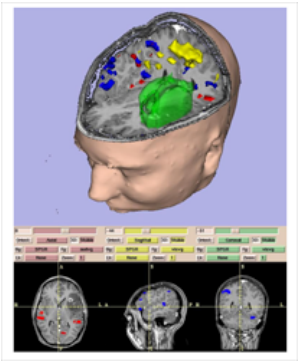


Figure 2.15: AstroMed view of segmented parts of the human brain [38]

Figure 2.16: The Op3D system being used in the operating theatre [9]

The Manchester Visualization Centre (http://www.rcs.manchester.ac.uk/) has several projects in medical imaging [9]. Their endovascular surgical planning software provides various visualizations of volumetric images and surface extraction techniques such as the Marching Cubes algorithm. They also make use of recent advances in hardware, such as the VolumePro PCI card from TeraRecon, that allow

volumetric images to be rendered with an increased performance compared to software alone. Their Op3D system (Figure 2.16) makes use of texture mapping hardware on a high performance computer to allow interaction with 3D volume renderings in the operating theatre.

ITK-SNAP (www.itk-snap.org) is an open-source project that allows easy segmentation of structures in medical images, as shown in Figure 2.17 [36]. It provides semi-automatic segmentation through the active contour method, as well as manual tracing and delineation of regions [39]. Segmentation and navigation of the image can be performed using three orthogonal planes (axial, sagittal, and coronal), and a simple user interface for selecting parameters. A 3D view also allows visualization of the surfaces of the segmented structures, and regions can be relabelled with a 3D cut-plane tool. As the name suggests, ITK is used to provide image processing features, and VTK is used for geometry construction and simplification.

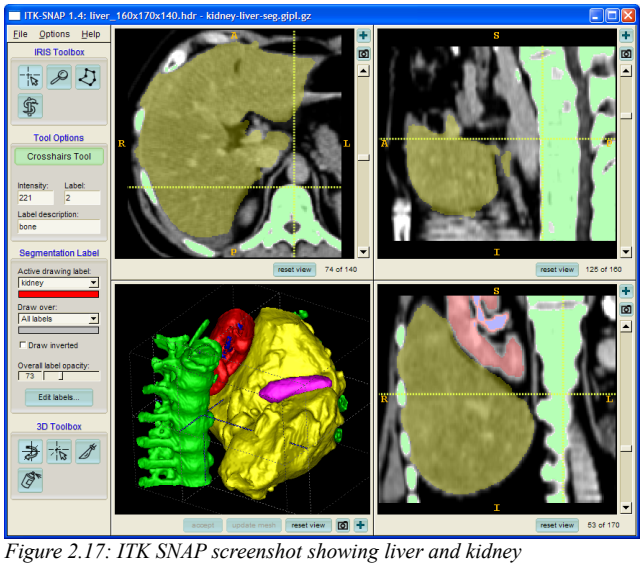


Figure 2.17: ITK SNAP screenshot showing liver and kidney segmentation [36]

Researchers at Shanghai Jiao Tong University, China have developed a system for osteosarcoma reconstruction using VTK, ITK, and FLTK for supporting functionalities [40]. The main functions of the system are image data acquirement, image registration and information fusion to combine several sources of data, segmentation to separate the tumour from the rest of the scene, 3D reconstruction of the surface of the tumour, display of the 2D and 3D images, and an interactive interface to examine and evaluate information. The system exploits the user's initial segmentation to suggest a segmentation based on the intensity patterns found in the different datasets. Surface rendering is accomplished by applying the Marching Cubes algorithm. The combination of the system's components provides doctors with the ability to segment, reconstruct, and visualize osteosarcoma tumours for diagnosis and planning procedures.

2.6 - Segmentation

The different regions in volumetric images need to be identified so that they can be visualized and manipulated individually. Segmentation is this process of labelling voxels according to the type of material or part of anatomy [2]. The user can perform segmentation manually by tracing the desired structures in each slice; however, this is very time consuming. There are various semi-automatic and completely automatic segmentation methods which give good results in less time and with less user effort. However, there are trade-offs between different segmentation methods in terms of automation, user interaction, and accuracy.

The simplest and fastest method of segmentation uses threshold values. If the value at a point is within defined threshold values, it is considered to be part of the corresponding object [2]. These threshold ranges can sometimes be determined automatically using histograms of the values in the image. Peaks and valleys in the histogram can be used to locate local clusters in the image [41].

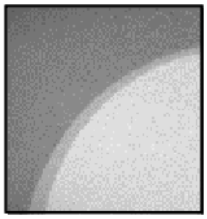


Figure 2.18: Input image to be segmented [41]

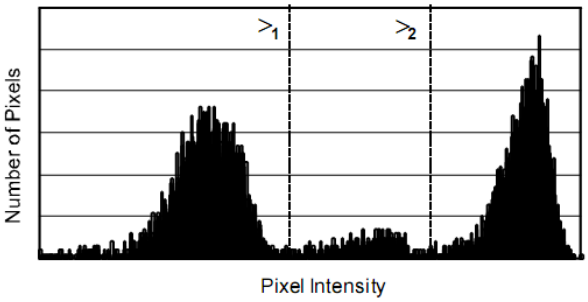


Figure 2.19: Histogram of the values in the image; the two valleys represent the borders between the regions [41]

Figure 2.20: Resulting segmented regions [41]

Unfortunately, identifying significant peaks and valleys may be difficult.

Figure 2.21: Input image to be segmented [41]

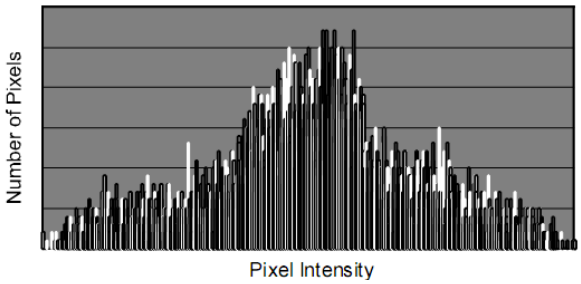


Figure 2.22: Histogram of the values in the image; there are no defined valleys [41]

The threshold values can also be user selected with the result updated in real time. However, the accuracy of the segmentation depends on the image and structures being segmented. If the structures cannot be easily divided by value, then parts of objects may become incorrectly labelled. Noisy images contribute further to this problem since the values of an object will vary even among adjacent points.

Objects in an image can be segmented easily if the boundaries between them can be identified. The results of an edge detection algorithm can be used as the boundaries of these objects [41]. However, edge detection usually leaves gaps in edges, whereas completely closed regions are necessary for segmentation. The found edges may also not actually correspond to object boundaries because of noise artefacts or features within an object, and smooth boundaries may not be found at all. Furthermore, the user has little control over the edges detected, leaving little room for improvement. Edge detection algorithms are often more useful to provide a heuristic in more advanced segmentation algorithms.

Region growing algorithms begin with a set of markers or seeds placed in an image [42]. These seeds then grow into regions until all pixels in the image are assigned to a region. Figures 2.23 and 2.24 show an example of region growing from a user specified seed region.

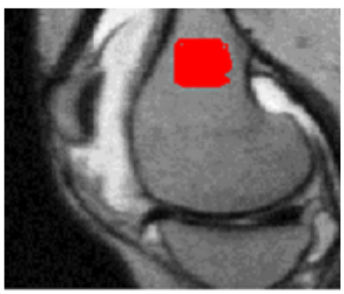


Figure 2.23: User specified seed region [43]

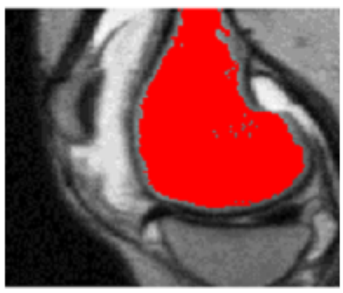


Figure 2.24: Final segmentation after growing the seed region [43]

Several heuristics can be used to determine how a seed grows. One such homogeneity heuristic compares the intensities of pixels around the border of a region with the mean intensity of that region to determine if the pixel should be added [44]. In some methods, the regions compete over pixels so that the best matching region wins; however, this requires a seed for every region in the image [43]. Others, such as the watershed transform [45] use the gradient of the image: adjacent pixels which are “down hill” are included in the same region, much like water pouring down to the lowest point, the local minimum.

However, the final regions are highly dependent on the initial seeds, and smooth boundaries are not guaranteed. To address these problems, constraints can be applied to the borders of regions [43]. Regularized region growing allows a region to be represented by a polygon, as shown in Figure 2.25. As the region grows, the corner points of the polygon move, which also causes other pixels to be included in the polygon region. Edges of the polygon are split as necessary when the corner points move too far apart. Regularized regions can prevent regions from growing through a small gap and provide smoother boundaries; however, the boundary between regions may leave many pixels unassigned because the polygon vertices do not lie at the same points. Figures 2.26 and 2.27 compare unregularized and regularized region growing.

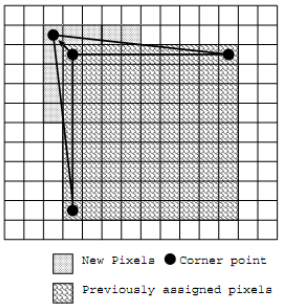


Figure 2.25: Regularized region growing [43]

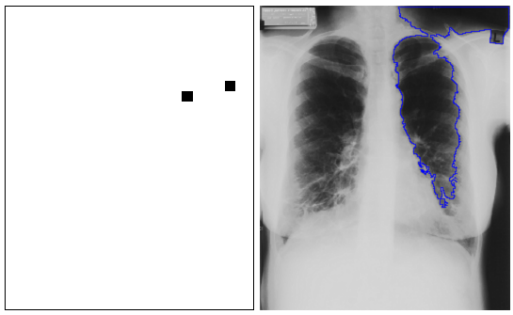


Figure 2.26: Seed points and results of unregularized region growing [43]

Figure 2.27: Results of regularized region growing [43]

The seeds used for region growing may be interactively placed by the user or sometimes automatically generated, and some algorithms allow the user to interactively place more seeds to improve the segmentation without completely reprocessing.

Active contours or "snakes" are another useful segmentation algorithm [46]. Rather than seed points, the user places a circle or other enclosed curve either within or around the desired object. The curves are then iteratively updated, subject to constraints, until the boundary segments the object from the rest of the image, similar to how a boa constrictor encircles and tightens around its prey. Figures 2.28 and 2.29 show examples of active contrours.

Figure 2.28: Simple active contour [46]

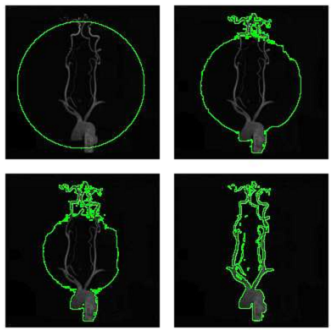


Figure 2.29: Active contour using level sets [46]

This algorithm is limited in use because the topology of the object must be known in advance, as a curve cannot change connectivity by splitting or joining with other curves. This constraint is well suited to large organs, such as the liver; however, blood vessels have a much more complex structure. To overcome theses difficulties, the active contour can be modelled as a level set of a higher dimensional surface, the extra dimension usually considered to be time [47]. A level set is defined by the set of points with the same function value. The value of the function for a given point and time is the distance from that point to the boundary of the curve at that time. The distance is positive if the point is outside the curve and negative if it is inside. These values are calculated for each pixel at each time iteration. Over time, the curve of each level set moves in the direction of its normal until the surface (zero level set) reaches the object's boundaries. A drawback to this algorithm is that once the algorithm begins, the user has no control over the segmentation and can only repeat the algorithm with another start curve.

An edge-stopping function can be defined so that the propagation of the curve halts where the

gradient of the image is high (where an edge is likely) and continues where the gradient is low (where an edge is not likely) [46]. The dependence on the image gradient is a disadvantage because not all edges are defined by high gradients, and the edge-stopping function may allow the curve to pass through a boundary. Furthermore, when images are smoothed to reduce noise, the edges are also smoothed, lessening the gradient. Figures 2.30 – 2.33 show examples of an image, its gradient image, edge-stopping function, and resulting segmentations.

Figure 2.30: MRI image of a knee [46]

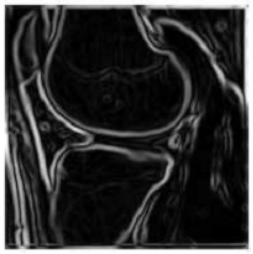


Figure 2.31: Gradient image [46]

Figure 2.32: Edge-stopping function [46]

Figure 2.33: Segmented femur [46]

Energy minimization provides an alternative to an edge-stopping function [48]. This method allows the length of the curve and the area of the region inside the curve to be minimized at the same time as a fitting energy is minimized. The fitting energy is made up of two parts: the sum of the differences between each pixel value inside the curve and the average value of all pixels inside the curve, and the sum of the differences between the outside pixels and average. Thus, the fitting-energy is minimized when the pixels inside the curve have about the same value and the pixels outside the curve have about the same value. This condition holds when the curve is on the boundary of the object. Unlike the edge-stopping method (Figure 2.34), the energy minimization method will not collapse on itself for an object with smooth contours (Figure 2.35).

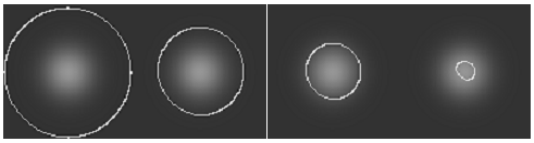


Figure 2.34: Edge-stopping method does not detect an object with a very smooth boundary [48]

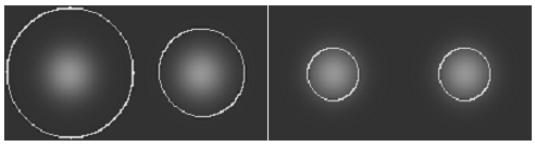


Figure 2.35: Energy minimization method can detect an object with a very smooth boundary [48]

Graph theory can also be applied to segmentation algorithms. Graph partitioning methods represent an image as an undirected graph with edges between every pair of pixels or small regions [49]. Each edge is given a weight depending on the similarity of the two pixels that it connects. Finding an optimal cut-set partitions the image graph into regions. Though graph partitioning can find a global optimum, the high degree of computation makes this infeasible. Restrictions and simplifications are made to improve computation, but also result in sub-optimal solutions. For example, the normalized cuts approach restricts pairwise similarities to local neighbourhoods. Several techniques exist for finding the optimum cut-set. A common technique is to find clusters of graph nodes which form the regions of pixels. Another method is to find cycles that minimize the cost function to form the closed contours of a region. Recent work has combined graph partitioning with the evolution of active contours to avoid neighbourhood size limitations and to add flexibility.

Hierarchical segmentation works similarly to graph partitioning in that it includes information about connected image regions with a tree structure [10]. This tree structure may be binary or n-ary, depending on the desired number of sub-regions per region. The segmentation may take a bottom-up (agglomerative) approach by clustering pixels together as children of a common root region, or it may take a top-down (divisive) approach by recursively sub-dividing the image into smaller regions. Divisive algorithms are more complex computationally due to their inherent lack of locality, so agglomerative algorithms are usually preferred. Regions are iteratively merged into a single region, with sub-region information stored in the hierarchy. The strength of an edge between two regions indicates how likely these sub-regions are to belong to the same region; this strength can be based on the difference between the average value of the two regions. Weaker edge strength means that the two sub-regions are more homogeneous and likely belong to the same region and should be merged first. Merging may take place

either locally or globally. Global merging considers the pair of regions separated by the globally weakest edge during each iteration. Local merging considers two regions at a time and selects a local edge which is weaker than any other edge involving either of these two regions. Local merging has the benefit of being computationally faster than global merging, but may not produce the globally optimal hierarchy. An advantage of hierarchical segmentation is that it allows the user to interactively mark more points as inside or outside the desired object and improve the segmentation without fully executing the algorithm again.

Applying known models to segmentation methods is often able to produce improved results [44]. Knowledge about the shape and intensities of landmark features in an image can be parameterized. This parameterization can be used with image registration algorithms to find the landmarks in images. Large scale, easily obtainable structures are often useful as a basis to guide the segmentation of smaller structures. A drawback to using parameterization and image registration is that training sets are often required to encode the shape and intensity variations. Further, modelling organs can be especially difficult because of differences in anatomy, pathology, and the deformability of soft tissues. More information for segmentation can also be obtained over time [50]. By combining temporal information with an *a priori* model about movements and volumes, more accurate segmentations can be performed on some parts of the anatomy, such as the heart (Figures 2.36 and 2.37).

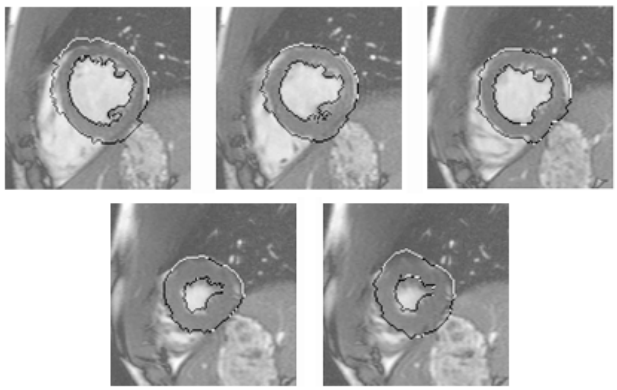


Figure 2.36: Segmentation of the heart in 3D+time [50]

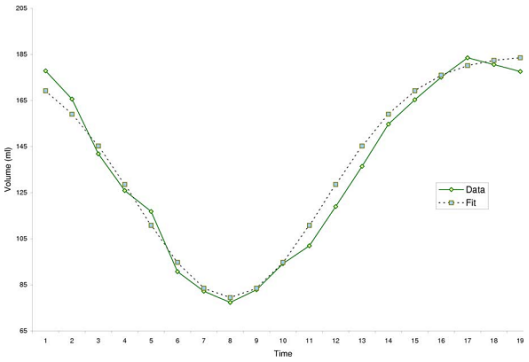


Figure 2.37: Model of the volume of the heart at different times and the fit of the segmentation [50]

Another method of improving segmentation results is by using an adaptive approach [10]. Adaptive segmentation methods first perform a segmentation algorithm on a scaled down version of the image and then update the results by considering larger scaled versions. This approach can provide better segmentations of some objects because edges will have larger gradients and noise will have less effect on the scaled down images.

2.6.1 Skeletonisation

Because of their tubular shape, blood vessels can be represented compactly by a skeleton. A skeleton can be a curve, such as the centreline, or it can be a collection of points [51]. Skeletons efficiently and compactly describe the shape of the original object, and the object or an approximation may be reconstructed from the skeleton. Skeletons have several uses, including automatic navigation, compression, shape description and abstraction, tracking, and animation control. Centreline curves are particularly useful for representing blood vessels. This centreline can be used for automatic navigation and surgical path planning with a virtual camera following the line to allow exploration of the vascular structure. Skeletonization is the process of reducing an object in an image to a skeleton that largely preserves the extent and connectivity of the original region. Topological thinning, distance transform, and Voronoi methods are the most commonly used for skeletonization.

Topological thinning is a fast skeletonization algorithm based on the topological properties of the object in the image [52]. Simple points are selected that when removed will not change the topology by disconnecting neighbouring points [53]. End points may be chosen by the user to remain unchanged and avoid excessive thinning. Without this constraint, a cuboid might be thinned to a single point [51]. This problem can also be avoided by removing one layer of points at a time until removing a layer would

break the connectivity constraint [54]. Thus, topological thinning guarantees connected skeletons which are useful for centreline extraction; however, reconstruction of the original object may not be possible because the distance from a skeleton point to the boundary of the object is unknown [51]. The skeletons of two unique objects, such as a box and a rounded box, may have the same skeleton, making the comparison of these skeletons less useful. Topological thinning may also produce centrelines with undesired loops within complex objects [54] which must be removed to produce the final centreline, as shown in Figure 2.38.

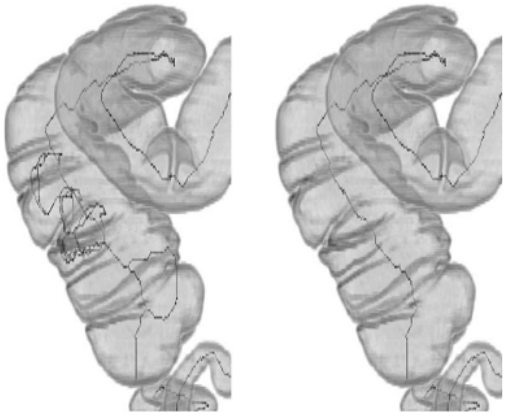


Figure 2.38: Results of applying topological thinning on the colon, and the results after removing loops in the centreline [54]

Distance transform methods [51] overcome some of the problems of topological thinning by assigning each point within an object the distance to the closest boundary point, as shown in Figure 2.39. By removing points with lower distances, thinning is performed and a skeleton is formed. In addition, the object can be reconstructed by using circles or spheres centred at each point with radius equal to the distance transform. More points can be removed if their corresponding circles lie entirely within the circle of any other point. Figure 2.40 shows the skeleton generated from a segmented trachea dataset.

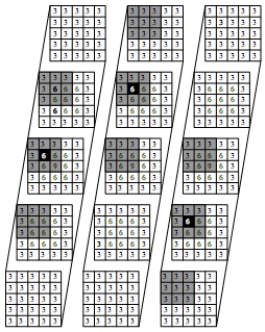


Figure 2.39: Distance transform values [51]

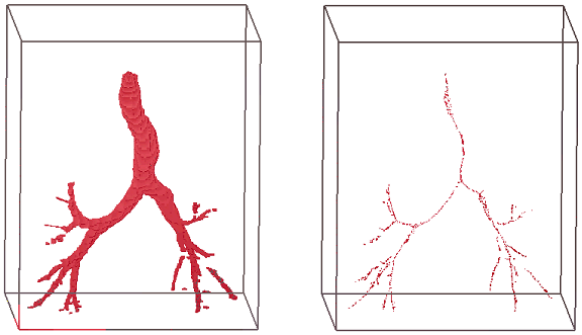


Figure 2.40: Trachea and its skeleton found using the distance transform method [51]

Unlike topological thinning, the resulting set of points does not guarantee a connected skeletal representation; however a connected centreline can be extracted [51]. If the user chooses endpoints from the skeletal points, then a simple midpoint subdivision algorithm can generate the centreline. The subdivision algorithm recursively locates the skeletal point closest to the midpoint of the current endpoints. This recursion continues between these new points until the distance between the current endpoints is less than a fineness parameter, which gives the desired distance between each pair of adjacent points that make up the centreline. Decreasing the fineness parameter decreases the length of each centreline segment, allowing the resolution of the curve to be controlled. A disadvantage of using the points closest to the midpoint is that small branches in the skeleton could affect the centreline, but this can be improved by taking into account the distance transform values. By selecting the closest point with the greatest distance transform value, a point close to the centre of the vessel is chosen. Distance transform methods allow multiple centrelines to be generated quickly without regenerating the skeletal points, and the user has the choice over which branches to create centrelines for by specifying the endpoints. The

centreline generated from the trachea skeleton is shown in Figure 2.41.

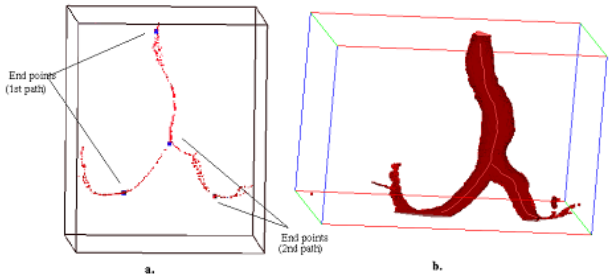


Figure 2.41: Generating the centreline from the skeletal points [51]

Voronoi methods provide another set of skeletonization algorithms based on graph theory [51]. The methods employ the Voronoi Diagram to find the centreline. When considering a set of points in a plane, the Voronoi polygon of one of the points is the polygon enclosing all points that are closer to that point than any other point in the plane. The Voronoi Diagram is then the collection of these polygons. The centre, or medial axis, of an object is equidistant from at least two points on the boundary. A Voronoi Diagram of the boundary points will give this medial axis: the Voronoi polygons of boundary points on one side of the object will meet the polygons of the points on the other side in the middle of the object. The Voronoi Diagram can also be extended into three dimensions, where the resulting faces will be near the centre of the object, giving part of the medial axis. Because only boundary points are used, Voronoi methods are best suited for polygonally defined objects, rather than volumetric data. Voronoi methods also lack user interaction and control over the segmentation process.

Most skeletonization algorithms do not handle noisy images well: the noise in images causes the shapes to appear sparse and unconnected. The simplest method to handle noisy images is to apply a noise cancelling filter before skeletonization [9]. The filter can be either isotropic which does not take direction into account or anisotropic which does. Anisotropic filters are best for vessel segmentation because they can enhance the vessels' contours at the same time as reducing the noise. Another method for handling these images is based on approximating the principal curve of the shape distribution [55]. In statistics, a principal curve is a smooth curve which passes through the middle of a data distribution or point cloud. The self-organizing map algorithm is used to find an approximation of the principal curve. A self-organizing map is a neural network such that the neurons specialize through competitive learning to represent different types of inputs. Neighbouring neurons are also affected so that they will group together representing similar inputs. By representing the pixels in an image with neurons, the pixels will group together to form neighbourhood relationships. Using a minimum spanning tree to define these relationships, a skeleton of the object is produced, like that in Figure 2.42. This method generates a useful stick figure-like structure on noisy images where other algorithms may fail; however, the original object cannot be reconstructed from the resulting skeleton.

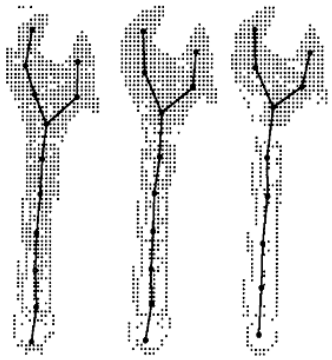


Figure 2.42: Skeletonization of sparse objects [55]

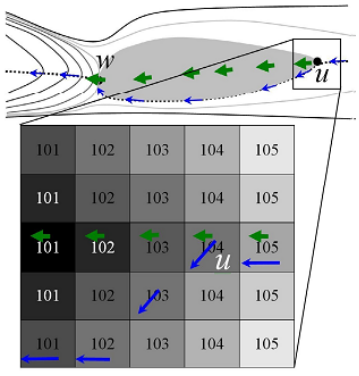


Figure 2.43: Fuzzy connectedness tree [56]

Fuzzy connectedness tree algorithms work in a similar way, but cannot handle sparse objects [56]. In this tree, each pixel points to its strongest neighbour, based on intensity value. This method assumes that higher intensities are found in the centre of the vessel and so pixels will point to higher intensities. The

centreline can then be followed from endpoint to endpoint; approximate boundaries can also be found by following pixels with similar intensity values around the object. Figure 2.43 shows the connections which form the centreline and boundary of the region. However, this algorithm will fail when the intensity assumption does not hold, which occurs with noise in the image or with structures for which the relative intensity values of the centre and boundary are not uniform or smooth.

Images of large dimensions are often encountered in medical imaging: a dataset of 600 slices of 512 x 512 pixels is 300MB of data if each pixel has an unsigned short value. Large volumetric datasets cannot be loaded at once in standard computer memory and must be skeletonized block by block and the results combined [57]. Care must be taken when a vessel runs along the border of a block, otherwise the skeleton of the vessel will be forced to the border because information about the other side is unknown. If maximal spheres around the points are considered, then a point can be deleted from the skeleton if its sphere is entirely included within the block. A point in the block on the other side of the boundary will remain because the centre of the vessel is on that side and the maximal sphere will stretch across the boundary. Alternatively, the blocks could be divided with some overlap between them and the skeletonization results then merged on the overlapping regions.

Blood vessels often prove difficult to segment because of their thin tubular structure and tree-like branches. Therefore, various segmentation methods have been proposed specifically for tubular structures. A simple method is the use of second order Gaussian, or Hessian, line filters to find lines of a specific thickness [58]. By changing the filter parameters in different passes, different line thicknesses can be segmented. Figure 2.44 shows the results of applying line filters to segment different sized lines.

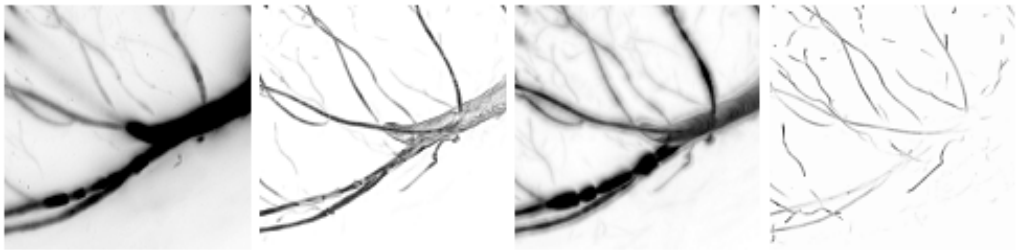


Figure 2.44: Results of applying various line filters on an artery and smaller branching arterioles [58]

The Hessian matrix can be used to calculate a vesselness measure for each voxel in a volumetric image [59]. The vesselness describes how likely the voxel is part of a tubular structure, such as a blood vessel. The Hessian matrix describes the second-order structure of local intensity variations around each point, and its eigenvalues and eigenvectors provide the amounts and directions of variation. Because blood vessels are long, tubular structures, they have significant variation in only two directions. If a vessel is brighter than the surrounding regions, it will have two large negative eigenvalues that correspond to the variation within the cross section. Since the cross section should be very similar to a circle, these eigenvalues will be very similar in value. The third eigenvalue describes the variation along the length of the vessel, which is usually negligible and close to 0. Other shapes have different eigenvalues, corresponding to different amounts of variation in each direction. Plate-like structures have two eigenvalues close to zero and one large eigenvalue, while spherical structures have three large eigenvalues. If the structures are darker than the surrounding region, then the large eigenvalues will be positive; and if they are brighter, then the eigenvalues will be negative. Thus, when filtering the image for vessels, these other types of structures can be excluded. The vesselness measure is then calculated by applying weights to different combinations of eigenvalues. For example, if three eigenvalues are similar in value, then the structure is more blob-like and the vesselness will be lower than if one of the eigenvalues were near zero. By calculating the vesselness for each voxel in the image, this filter provides a fast segmentation method which requires little input from the user.

As with any segmentation, various heuristics and *a priori* knowledge can also provide improvements. One such source of *a priori* knowledge includes information about the contrast agents used during imaging. Dynamic Contrast-Enhanced Magnetic Resonance Imaging allows a sequence of images to be acquired over time as a contrast agent is injected to enhance specific tissues [50]. Measurement of this enhancement allows diagnosis and can also be used with heuristics to find and segment the enhanced tissue. Other methods use knowledge of intensity changes on boundaries of

specific structures [60], and Bayesian tracking can be used to find the best match of shape and appearance models to image data [61]. Because of the wide variety of segmentation methods available, standard practise is to start with the simplest methods and to move on to more advanced methods if the results are not adequate [2]. More information, if available, can also be supplied to improve segmentation, and algorithms can be modified to better suit specific purposes and structures if necessary. Once the desired segmentation is complete, it can then be used to generate surface meshes of the objects.

2.7 - Mesh Generation

The surfaces of complex objects can be represented by compact triangular meshes. The efficiency that meshes provide for both visualization and simulation is necessary to satisfy the real-time constraint of virtual reality simulators. Unlike volumetric data, triangular meshes can be efficiently rendered by common graphics hardware, with textures providing a realistic appearance [2]. They can also be easily deformed and animated by simply moving the vertices. Meshes lend themselves well to a mass-spring system which can be used to model deformable soft tissue: the vertices act as masses and the edges act as springs. These meshes can also be used with the finite element method to find approximate solutions of partial differential equations in simulations. However, surface meshes alone do not contain volume information, so a mesh structure must also be created within the surface to represent the volume of the object [62]. The surface mesh can be generated from isovalue layers or segmented objects in the image, and the internal mesh can then be generated from the surface mesh.

The marching cubes algorithm [63] is the most common method of generating a triangular surface mesh from volumetric data. Each voxel is processed independently, and linear interpolation along the edges of the voxel gives the vertices of the surface approximation. The surface fragments for each voxel is determined from a set of cases which form a lookup table. Lorensen and Cline's Marching Cubes algorithm uses a lookup table of 256 possible polygon configurations, which can be obtained by reflections and symmetrical rotations of 15 unique cases (Figure 2.45). However, this algorithm has flaws in certain cases [64].

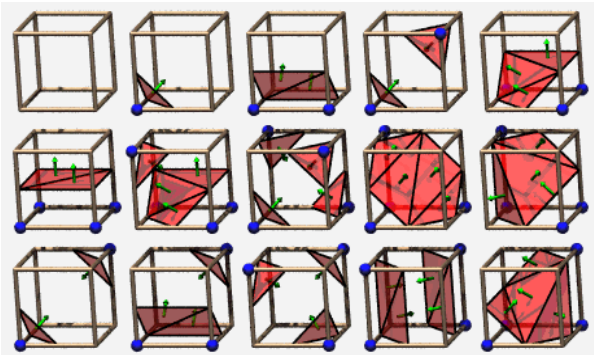


Figure 2.45: 15 unique triangulation cases for the original Marching Cubes algorithm [65]

Variants of the marching cubes algorithm have been proposed because of problems with topological inconsistency, cracks in adaptive resolution, and inability to preserve sharp features [64]. Topological inconsistency is due to ambiguities that arise when there are more than one possible assignment for a case in the lookup table. If ambiguities are not resolved appropriately, the surface could have holes. Adaptive methods reduce the number of resulting triangles in the surface by applying marching cubes to a grid which has different sized sells; however, different resolutions between cells often cause cracks in the mesh that must be patched. The original marching cubes algorithm also assumes the surface is smooth and does not preserve sharp edges and corners. One method of preserving sharp edges is to consider the intersection of tangent planes [66]; however, this introduces inter-cell dependency and adds to computation. Holes have been corrected with different possible triangulation cases and by having more than one possible triangulation for problematic cases. Bilinear interpolation can be used to decide on ambiguous faces by determining if two opposing vertices should be connected or separated. Trilinear interpolation on the interior of voxels further solves ambiguities and preserves the same connectivity or separation of vertices as given by trilinear interpolation of the volume data itself.

However, adding further interpolation adds more unique cases to the lookup table.

A cubical marching squares algorithm has been proposed to solve these problems in a much simpler manner [66]. This algorithm reduces the three-dimensional problem into a two-dimensional one. Each cube can be unfolded into six squares which are processed independently. Face ambiguities can be resolved easily in 2D and then the resulting components triangulated to generate the final surface. Because the algorithm works in 2D, the lookup table is much smaller for marching squares than for any marching cubes algorithm. Figure 2.46 shows all cases for the marching squares algorithm.

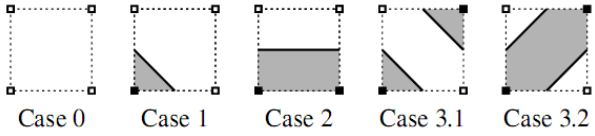


Figure 2.46: Triangulation cases for cubical marching squares [66]

2.7.1 Simplification

Because each voxel is triangularized independently with marching cubes algorithms, the number of triangles in the final surface mesh is at least as large as the number of voxels. Such a large number of triangles is often unnecessary and can be reduced through a process called decimation while preserving the shape and topology as much as possible [67]. Decimation algorithms first classify vertices to determine good candidates for deletion. A vertex may be a simple vertex surrounded by adjacent triangles, a boundary vertex on the boundary of the mesh, an interior vertex which is on a feature edge of the object, or a corner vertex which is on two feature edges. A feature edge is an edge with adjacent triangles whose normals are greater than a user specified angle apart. A vertex may also be complex if two surfaces intersect at that point; however, marching cubes algorithms do not produce complex vertices. Each vertex classification is illustrated in Figure 2.47.

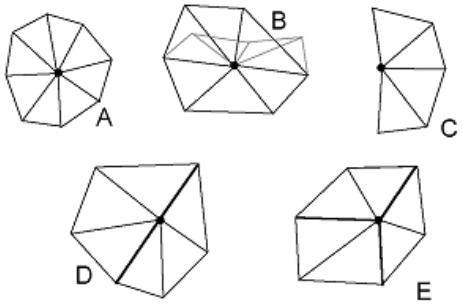


Figure 2.47: A: Simple vertex,
B: Complex vertex
C: Boundary vertex, D: Interior vertex
E: Corner Vertex [68]

In general, a vertex is removed if the distance between it and the average plane of its adjacent vertices is greater than a user specified error value [68]. For boundary and interior vertices, the distance from the vertex to the edge that would be formed by removing the vertex is used instead. After removing a vertex, the resulting hole must then be filled by triangulation. All vertices that match the distance criteria can be removed, or a priority queue can be used to remove vertices with larger distances first and then stop when a specified amount of reduction has been completed (Figure 2.48). The decimation criteria may also be modified to use different error measures other than the distance to the average plane when determining whether a vertex should be removed or not. The quality of the resulting decimated mesh is limited because new vertices cannot be created to ensure uniformly sized triangles.

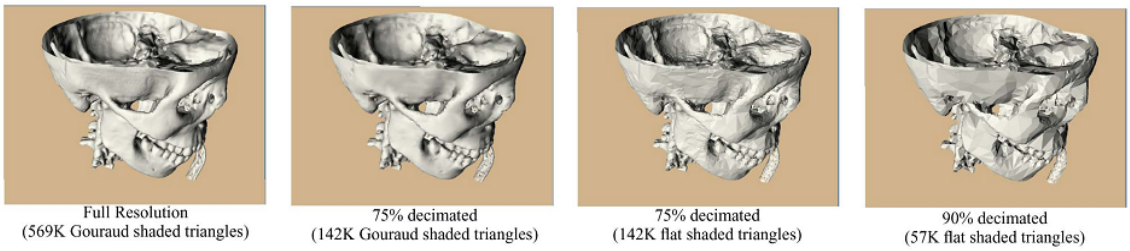


Figure 2.48: Results of decimation with different numbers of removed triangles [68]

Edge contraction [69] provides an alternative to decimation for simplifying triangular meshes. This algorithm repeatedly selects the edge which contributes the least to the shape of the mesh and removes it. The edge is collapsed, merging the two end point vertices into a single new vertex, as illustrated in Figure 2.49. The two triangles that were adjacent to the edge also become degenerate when the points are merged, so they are removed from the mesh. The remaining edges and triangles which were adjacent to the edge are modified so that they use the new vertex rather than the two original vertices. To determine which edges of the mesh should be collapsed, the algorithm ranks them in a priority queue by their cost. The cost of an edge defines its contribution to the shape of the mesh, so that removing edges with lower cost will have less effect. The algorithm then collapses the edges in order from least to greatest cost. After each edge is removed, the costs of the adjacent affected edges need to be recalculated. The collapse iterations continue until a certain reduction has been met, providing the simplified mesh.

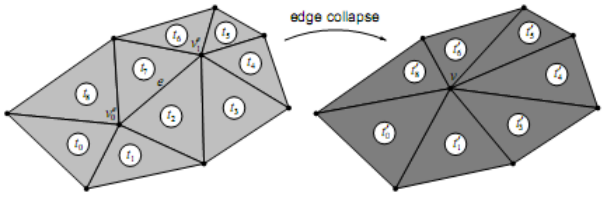


Figure 2.49: A single iteration of edge contraction [69]

The edge contraction algorithm can be modified to suit different structures and quality criteria by using different edge cost metrics and by changing where the new vertex is placed. A simple edge cost metric is the length of the edge, since short edges generally affect the mesh less than large edges. This metric also ensures that the resulting triangles are not long and narrow, helping to improve the quality of the mesh. Other cost metrics are based on boundary and volume optimization. The placement of the new vertex can also be chosen to preserve properties of the original mesh. The vertex can be simply placed at the midpoint of the collapsed edge, but this is not optimal. By choosing different vertex positions, the volume and boundaries of the mesh can be preserved. The shape of the resulting triangles can also be optimized to provide a better quality mesh.

Another method for generating a surface from volumetric data makes use of Delaunay triangulation [70]. The Delaunay triangulation of a set of points is the dual graph of the Voronoi tessellation. In two dimensions, it is a triangulation such that no point is inside the circumcircle, or bounding circle, of any triangle. In three dimensions, the Delaunay triangulation is a triangulation such that no point is inside the circumsphere of any triangle. A surface mesh can be generated by sampling points on the surface of the object and performing a Delaunay triangulation of these points. A seed facet or random points can be used to give the initial samples, and further samples are taken, pushing the boundary edge around the object until the entire surface has been generated (Figure 2.50).

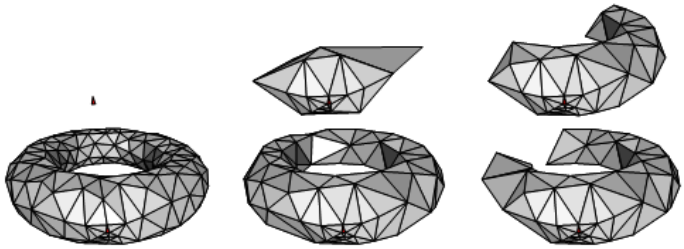


Figure 2.50: Meshing using the Delaunay triangulation of points sampled on the surface of a torus [70]

To ensure that Delaunay conditions are met, if a sample falls within the sphere with diameter equal to the length of an edge, then the sample becomes the midpoint of that edge. The desired distance between samples can be specified by the user to generate different resolution meshes, and the sampling distance can also vary according to curvature or other parameters to improve the generated mesh. This ability to control sampling is useful for generating quality meshes for use in simulations and is illustrated in Figure 2.51.

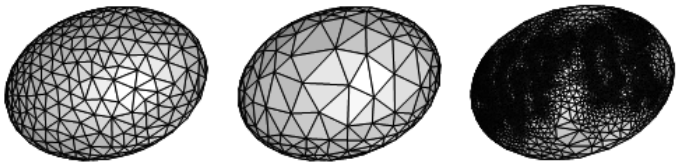


Figure 2.51: Various size criteria: uniform, curvature-adapted, and customized [70]

This method is not restricted to volumetric data, but it can also be used to re-sample an existing mesh; this ability makes it a useful alternative to decimation. However, this triangulation assumes the object to have a closed surface, unlike marching cubes [62].

2.7.2 Smoothing

The output from mesh generation algorithms may not be as smooth as the true surface of the object. For example, noise may cause the surface to have many bumps which must be smoothed to better represent the actual anatomy. Smoothing can also help improve the quality of decimated meshes by making the triangle vertices more regularly spaced like equilateral triangles. A common smoothing algorithm is provided by the Laplacian operator [71]. This method iterates through each vertex, moving it closer to the average plane of its neighbouring vertices (Figure 2.52). Thus, bumps and sharp edges in the mesh will be reduced, and the overall surface will become smoother (Figure 2.53).

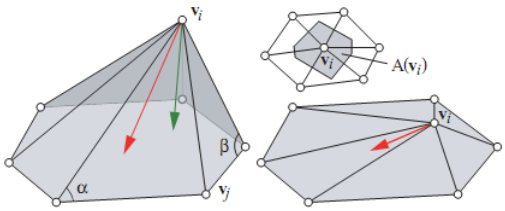


Figure 2.52: Moving the vertex towards the average plane to smooth the mesh [71]

The amount of smoothing is controlled by the number of iterations performed and the amount by which each vertex is moved towards the average plane, called the relaxation factor [71]. Large relaxation factors will cause the vertices to move larger distances, which may cause the smoothing to become unstable. If vertices are moved too far, then they may move in the opposite direction in the next smoothing iteration, causing parts of the mesh to expand and shrink out of control. Thus, small relaxation factors are typically used with a large number of iterations. Although Laplacian smoothing is very simple and fast, shrinkage of the mesh towards the centroid is inevitable. Shrinkage is especially noticeable on thin objects, making this method unsuitable for meshes of blood vessels.

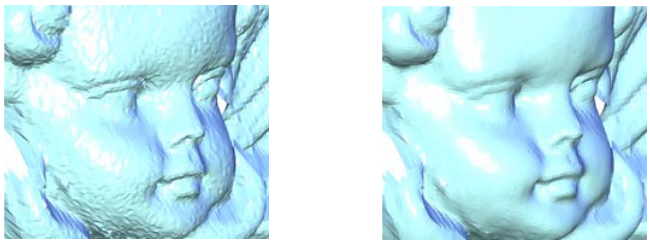


Figure 2.53: Mesh before (left) and after Laplacian smoothing (right). Notice the bumps have been reduced greatly, providing a much smoother surface.

Another smoothing algorithm makes use of standard signal processing methods [72]. By performing Fourier analysis on geometry, the frequency distribution can be determined and used to

smooth the mesh. In images, high frequencies correspond to high variation in intensity, such as caused by noise, while large regions with little intensity variation have low frequencies. In a mesh, high frequencies are apparent as bumps and spikes, while smooth regions have low frequencies. Just as a low-pass filter removes the high-frequencies and noise from an image, a low-pass filter on geometry will provide smoothing. Unlike Laplacian smoothing, this method only filters frequencies above a pass-band, greatly reducing the shrinkage of the mesh. Although a complete Fourier analysis would require too much computation to perform on a complex mesh, an approximation can be performed by applying two consecutive steps of Laplacian smoothing. The first step involves a standard smoothing pass with a positive relaxation factor, but the second pass uses a negative relaxation factor to grow the mesh, reducing shrinkage. The pass band can be used to chose relaxation factors that ensure a stable and fast smoothing filter.

2.7.3 Tetrahedralisation

In order to create the inner mesh of an object to represent its volume, 3D primitives such as tetrahedra must be generated within the surface mesh, separating the volume into small cells. Constrained Delaunay tetrahedralization is a process of generating these tetrahedra that are constrained by the surface mesh [73]. This process is similar to Delaunay triangulation except the points are sampled within the surface and tetrahedra are formed rather than triangles. Like before, rules are required to handle encroachments while sampling these points. If a point encroaches a subsegment, then the midpoint is used, as in Delaunay triangulation. If a subface is encroached, then the circumcentre is used. If a tetrahedron has a radius-edge ratio larger than desired, making it too thin, then it is replaced by its circumcentre. Overall, this method forms a good quality mesh; however, poor quality tetrahedra may still occur near small surface mesh angles. The mesh can also be further refined by adding new points that satisfy user-specified size constraints on the tetrahedra [74]. These size constraints control the volumes of the tetrahedra and the resolution of the inner mesh to form a tetrahedralization that can efficiently represent a deformable object in simulations (Figure 2.54).

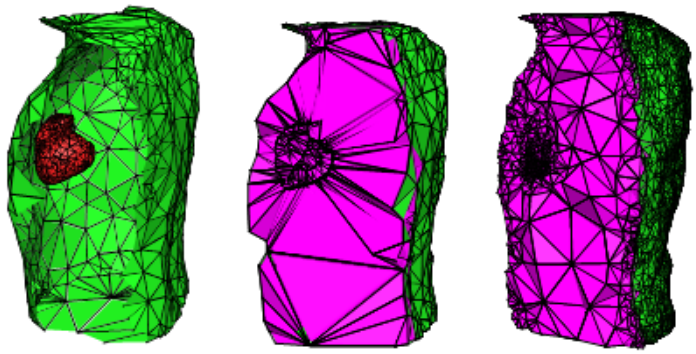


Figure 2.54: Original surface mesh, constrained Delaunay tetrahedralization, and the refined tetrahedralization [73]

2.8 - Toolkits and Libraries

Several toolkits and libraries provide open-source, cross-platform support for graphical user interface (GUI), image and geometry processing, and visualization functionality. Toolkits such as FLTK, ITK, and VTK can be used together to combine these functionalities into a single application. Other special-purpose libraries, including CGAL and TetGen, provide further advanced geometry algorithms. By using the CMake build system, applications using these toolkits and libraries can be compiled easily in various compiler environments on several platforms.

The Fast Light Toolkit (FLTK), available from http://www.fltk.org/, provides modern GUI functionality with native OpenGL rendering support and built-in GLUT emulation [14]. The toolkit also includes the Fast Light User-Interface Designer (FLUID), an separate application which allows user interfaces to be quickly and easily designed via a simple WYSIWIG, drag-and-drop interface. FLUID exports the completed GUIs in C++ code to be integrated with an application's back-end. FLTK's libraries

are small and modular so that they can be statically linked, but they can also be used as shared libraries. FLTK is provided under the terms of the GNU Library Public License, Version 2 with exceptions that allow for static linking.

The Insight Toolkit (ITK), available from http://www.itk.org/, provides image input and output for a wide variety of file formats, along with many image filters and transformations [15]. It also includes leading-edge segmentation and registration algorithms. Most of ITK's algorithms are not limited only to two-dimensional images, but can also be used on images with three or more dimensions. The toolkit was started to support the Visible Human Project and was developed by six principal organizations: Kitware, GE Corporate R&D, Insightful, UNC Chapel Hill, University of Utah, and University of Pennsylvania. ITK is provided under an open-source BSD license that allows unrestricted use, including use in commercial products.

The Visualization ToolKit (VTK), available from http://www.vtk.org/, provides a wide variety of visualization algorithms for scalar, vector, tensor, texture, and volumetric data [16]. It also includes advanced geometry algorithms and modelling techniques such as implicit modelling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation. Several imaging methods have also been integrated to combine 2D and 3D graphics algorithms and data, but filters exist to connect ITK and VTK pipelines when further functionality is required. VTK's visualization pipeline consists of data objects, which represent information, and process objects, which perform operations on data [6]. There are several types of data objects, including polygonal and volumetric data. Process objects can be separated into three categories: sources, filters, and mappers. Sources create new data objects from files or basic primitives, such as spheres and boxes. Filters perform an algorithm on input data to produce output data, which may not necessarily be of the same type. Mappers terminate the pipeline, sending the data to the graphics subsystem for display or writing it to a file or network. By linking sources, filters, and mappers together, data can be transformed into a format to be visualized by a renderer. VTK has an open-source license that allows unrestricted use and redistribution with or without modification.

The Computational Geometry Algorithms Library (CGAL), available from http://www.cgal.org/, provides efficient and reliable geometric algorithms in a C++ library [17]. It offers many data structures and algorithms including triangulations, Voronoi diagrams, Boolean operations, mesh generation, and other geometry methods. Parts of CGAL are provided under the less restrictive LPGL license while some parts are under a QPL license. The QPL license protects the higher-level algorithms so that any program using these parts must also be released under an open-source license or a commercial license must be obtained.

TetGen, available from http://tetgen.berlios.de/, is a small C++ library designed specifically to generate quality tetrahedral meshes [18]. It can also create Delaunay tetrahedralizations, Voronoi diagrams, and constrained Delaunay tetrahedralizations. The resulting tetrahedral meshes are suitable for solving partial differential equations by finite element or finite volume methods. TetGen is free for research and non-commercial uses.

Cross-Platform Make (CMake), available from http://www.cmake.org/, is a cross-platform, open-source build system [19]. It allows the generation of native build files, such as makefiles for Unix systems and project workspaces for Microsoft Visual Studio. CMake has the ability to find installed include files, libraries, and executables required for building and also provides a user interface to control the build process. It can also test the chosen compiler for supported features that may be necessary for a successful build. CMake was developed by Kitware as part of the ITK project, and it is provided under an open-source BSD license, which allows unrestricted redistribution and use.

3 - Design

Generating a compact, efficient representation of anatomy from patient datasets requires many steps, each building upon the results of previous steps. The sequence of algorithms for pre-processing, segmentation, skeletonisation, and mesh generation forms a long pipeline with many parameters and much user interaction. Because of the large amount of data and algorithms involved, separation of components was crucial to the design. Thus at a top level, a Model-View-Controller architecture was chosen. The Model stores the required data and performs the sequence of algorithms. The View displays the data to the user and allows interaction through a graphical user interface. The Controller handles the connection between the Model and View, sending user specified parameters from the View to the Model, and updating the View with the latest data from the Model. This separation greatly decreases dependence between the components and allows easy modification of a single component without greatly affecting the others.

At a lower level, the algorithms themselves are separated into smaller components. Each algorithm is performed through a specific tool, separate from other tools. Each tool has a Model component, which stores the necessary data and parameters for the algorithm, and a View component, which displays the information to the user and provides the ability to modify parameters. Parameters and data are updated through the Controller. As the user applies different tools, the pipeline from the original patient dataset through to the final output is formed. Such separation of tools greatly simplifies integration of many algorithms with corresponding user interfaces into the same program.

3.1 - Model

The Model component's main responsibilities are storing the data and performing algorithms on this data. The Model must also coordinate with the View through the Controller to update parameters for algorithms and visualise the output. Several classes are required to achieve this functionality, and ITK and VTK provide the basis for algorithms, their data, and preparation for visualization. The main Model class provides a simple, but flexible interface to its tools and algorithms. It handles calls from the Controller to load patient datasets from files and initializes the pipeline of tools with this data. Methods are provided for setting the current tool, passing algorithm parameters to tools, and performing tool actions. The Model also handles calls to save the modified dataset to file, along with loading and saving geometry. The collection of currently viewable objects is easily accessible so that they can be visualized by the View. These viewable objects provide a visualization of the data while the user modifies parameters and applies Tools.

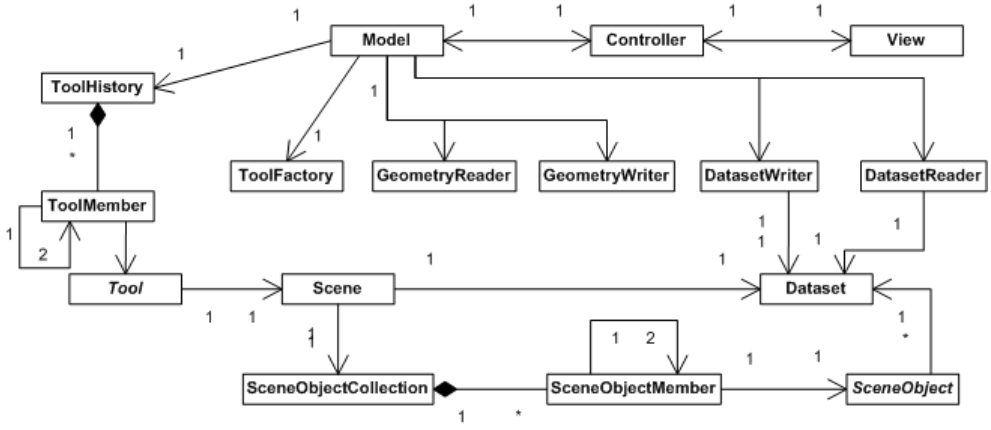


Figure 3.1: Simplified UML class diagram showing the relationships between the main components of the Model. See appendix for a more detailed diagram.

The main data type is the patient dataset, and most functionality regarding its storage is provided by the Dataset class. This class provides access to both ITK and VTK image data formats to allow importing datasets from various sources for easy use in algorithms from each toolkit. Information about the dataset, such as the intensity range, origin, extent, and voxel spacing can also be obtained from its

Dataset object. Because conversions between world coordinates and image voxel indices are required by several tools, helper methods are included to perform these conversions based on the origin, extent and spacing of the Dataset. A Dataset also defines the conversion from its stored scalar values to intensities displayed to the user. This correspondence between values and intensities is provided by a colour transfer function defined by a curve, as shown in Figure 3.2. The Contrast Tool uses methods of the Dataset to define the number and locations of control points on this curve, which is used by all slice and volume visualizations of the Dataset.

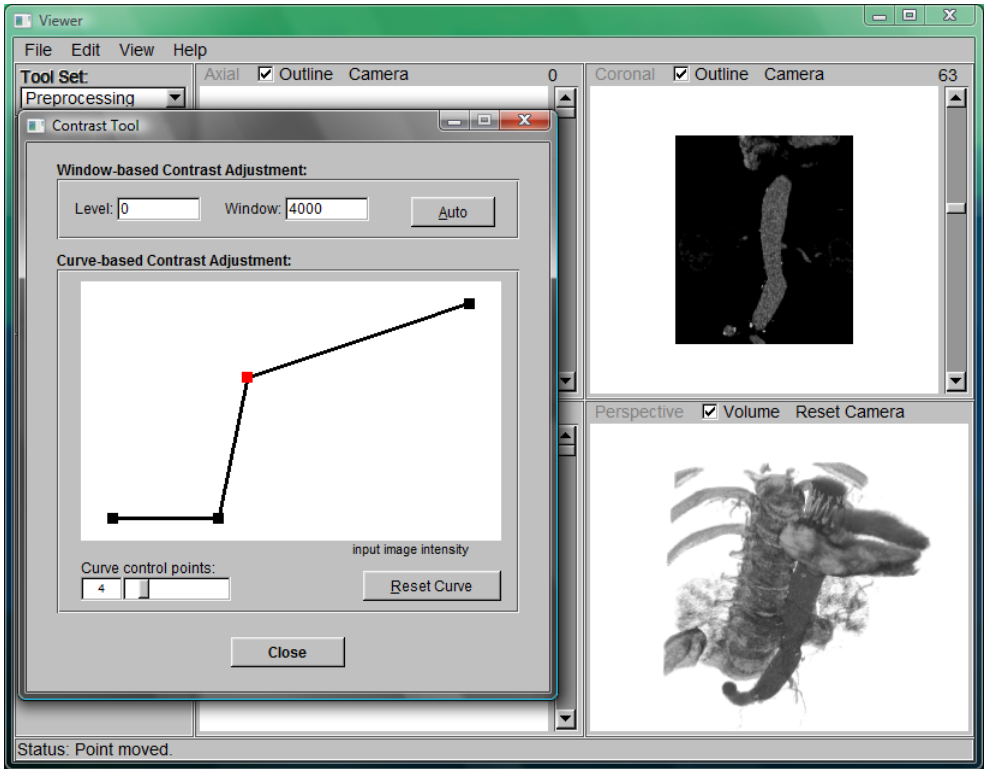


Figure 3.2: Contrast Tool window, on the left, used to adjust the intensity of a dataset. In this case, the lower intensities are removed so that the inner structures can be seen.

Dataset information is read from a file with the DatasetReader class and written to a file with the DatasetWriter class. The Model uses the reader to load the initial patient dataset and uses the writer to save modified datasets. Some tools and algorithms also require the ability to save and load images provided by these classes. For example, CGAL algorithms require the Dataset to first be written to a temporary file and then loaded into a CGAL image object for use. Supported dataset file formats include DICOM (.dcm), VTK Structured Points (.vtk), and those supported by ITK, such as Analyze (.hdr and .img), MetaImage (.mhd and .raw), and GIPL formats (.gipl). The file type can either be set to a specific type, or it can be determined by the extension of the file chosen to be read or written.

The other major data type is the geometry that provides a compact and efficient representation of anatomical structures. This geometry is stored in vtkPolyData objects for triangle meshes and vtkUnstructuredGrid objects for tetrahedral meshes. The GeometryReader class provides the ability to load geometry from files, while the GeometryWriter class allows it to be saved to files. The Model uses the reader when the user chooses to load already generated meshes from files and uses the writer to save the output from meshing algorithms. Some tools also make use of the reader and writer to perform algorithms with CGAL and TetGen: a temporary geometry file must be saved and then loaded by the library before performing the algorithm, and CGAL also saves its output to a file which must then be loaded again by the GeometryReader. Supported geometry file formats include VTK Poly Data (.vtk), Object File Format (.off), Virtual Reality Modelling Language (.vrml and .wrl), Wavefront Object (.wav and .obj), Stereo Lithography (.stl), Gmsh (.msh), and X3D (.x3d) formats. Again, the file type can be either specified or determined by the file extension when saving or loading files. Some tools require the saving and loading of geometry other than triangle meshes, such as points and lines, so other reader and writer classes provide the necessary functionality.

For visualization, these data types must be provided as inputs to SceneObjects. The SceneObject class represents any object that can be viewed by the user, and it provides basic functionality required by the specific types of objects, which are defined as sub-classes. A SceneObject stores the vtkProp objects that are rendered in each viewport, and sub-classes are responsible for setting up the required mappers and actors. SceneObjects also store viewing information such as position, colour, visibility, and whether or not the object can be selected by the user. The slice numbers that the user is currently viewing are also provided to a SceneObject so that the slices into the data can be generated. The methods used for setting the current slice numbers should be overridden to define how the slice is generated, and sub-classes can also override methods which set other viewing information to update the output accordingly. Of course, they may also define other methods necessary to set input data or to generate and modify the data as needed. Each sub-class of SceneObject has a unique constant integer identifier for quick identification of the type of object; these types include Volume, Mesh, Slice, Point, Line, Spline, TetMesh, SkeletalPoint, and CenterLine. These identifiers proves useful for sorting objects by type and determining the type of object selected by the user. Figures 3.3, 3.4, and 3.5 show examples of some of these objects.

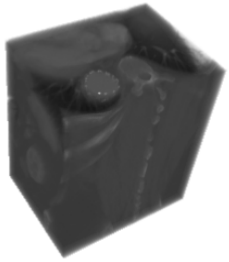


Figure 3.3: Volume of a patient dataset

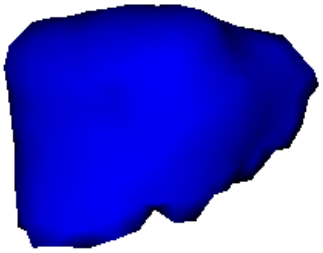


Figure 3.4: Mesh of a liver

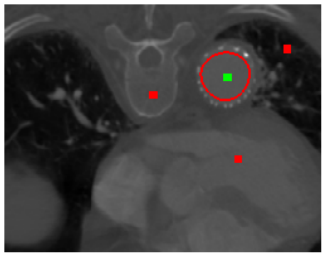


Figure 3.5: Spline and Points placed on an AxialSlice

The main sub-classes of SceneObject are Volume and Mesh. The Volume class allows for the visualization of the Dataset as a volumetric rendering, while Mesh allows the visualization of triangle mesh geometry. Several classes extend from the Mesh class to provide Slices, Points, Lines, Splines, and tetrahedral meshes (TetMesh). The Slice class generates texture-mapped, planar slices of the Dataset to give a 2D view of the data. Three sub-classes, the AxialSlice, CoronalSlice, and SagittalSlice classes, extend from Slice to provide the three traditional slice directions shown in the viewports.

Since many SceneObjects may be visible at the same time, a list of objects provided by the SceneObjectCollection class is necessary. This collection is a linked list of SceneObjectMembers, each of which has a corresponding SceneObject and a link to the next SceneObjectMember in the list. Besides methods for adding, removing, and accessing SceneObjects, the SceneObjectCollection class also provides helper methods for tools to highlight all stored objects as selected and set all objects or all of one type of object as selectable by the user. Tools can also obtain the SceneObject that corresponds to a user-selected vtkProp by calling a single method which will search the objects in the list for the one that uses the specified prop. The list of visible objects is stored as a SceneObjectCollection in the current Scene, and tools also use SceneObjectCollections when they need to store a list of objects.

The objects currently displayed to the user, along with current viewing options, are stored in a Scene. The current version of the Dataset is provided as an input to the Scene, which is responsible for preparing the Slice and Volume objects using this Dataset. The Scene also provides helper methods for setting the current slices, visibility of Mesh outlines in the slice viewports, and visibility of the Volume in the perspective viewport. Each time SceneObjects are added or removed to the Scene, it must be updated and the View must obtain the new collection of vtkProps for each viewport. Each time a new tool is applied, a new Scene is created to store the current state. Thus, a chain of Scenes represents the changing state of the Dataset and other objects as new tools are applied.

The Model component of each tool extends from the Tool class, which provides the basic functionality required by all tools. Each tool has a unique integer identifier, which in combination with a ToolFactory class allows easy creation of new tools. After the Model creates a new Tool, the current Scene is provided as input for the Tool to apply its algorithms. Tools generate an output Scene, which initially contains the same SceneObjects as the input, but may be modified as needed without affecting the original Scene. Each sub-class of Tool defines what algorithms to use and how the Scene should be

modified. The Tool class also has methods to allow parameters to these algorithms to be set and retrieved so that the Model can route this information from the Controller to the Tool. Specific Tool sub-classes must define a unique integer identifier for each parameter to be used when specifying which parameter is to be set or retrieved. Tools may also perform actions initiated by the user, and these actions also must have unique identifiers within the sub-class. These sub-classes are responsible for updating their displayed parameters and Scene by calling Controller methods. Mouse events such as pressed buttons, movements, and released buttons are also passed to the Tool to be handled as needed. Tools may load and save files, so methods are provided for the Model to forward the chosen file name and type. Finishing a Tool applies its final results to the Scene so that another Tool can be chosen and initialized on the output of the previous Tool. Tools may also be cancelled, resetting the Scene to its previous state and freeing the Tool's memory. Each sub-class of Tool can define versions of these methods for its specific purposes, providing the flexibility required for any algorithm. To minimise the complexity of tools, complex algorithms are defined as separate classes with their own inputs provided by the tool and outputs used as necessary.

The tools link together to form a pipeline which modifies the Scene from its initial state to its current state. The ToolHistory class stores the list of applied tools so that the user may navigate to previous tools, change parameters, and update results. The ToolHistory is defined as a doubly linked list for easy traversal to previous tools and back to the current tool. ToolMember objects store the tool and links to the previous and next ToolMember in the history. Besides adding and removing tools from the history, methods also allow the user to go back to a previous tool and go forward to the next tool. When moving to a previous tool, the tool is updated with an Undo method so that it can return to a non-finalized state and update its output. When moving to the next tool, the tool is similarly updated with the Redo method. When moving to different tools, adding tools, or removing tools, the history keeps track of the current tool. The ToolHistory is also responsible for updating the View so that the current tool is displayed.

3.2 - View

The View component sets up and manages the graphical user interface of the program. The main View class receives and displays updates from the Model, allowing windows and tools to be shown and displayed parameters and the Scene to be updated. Current Dataset properties such as the slices, image intensity range, and image extent are also passed to the user interface to be updated and used for the valid ranges of input fields and sliders. The graphical user interface was built from FTLK components using FLUID, allowing for easy design and modification. The main program window is defined by the MainWindow class, while other smaller classes provide supporting functionality and other windows.

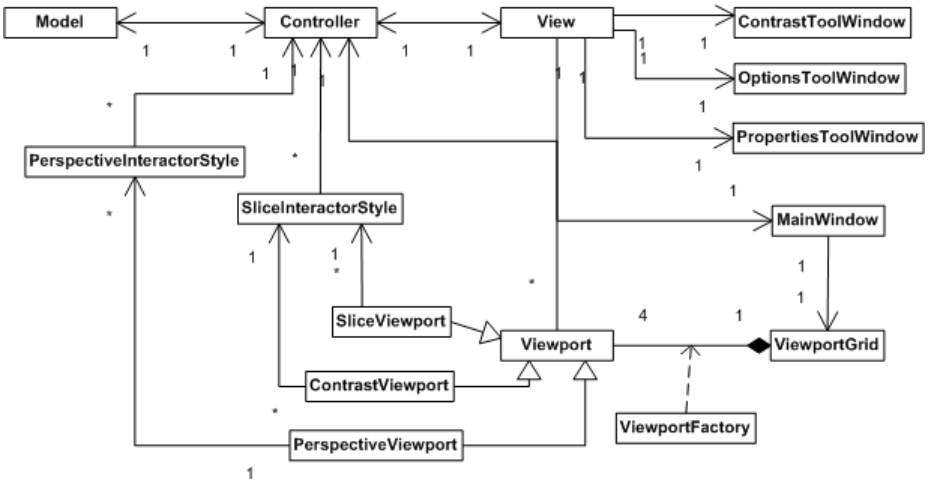


Figure 3.6: Simplified UML class diagram showing the relationships between the main components of the View. See appendix for a more detailed diagram.

The graphical user interface of the MainWindow class is made up of four resizable rendering viewports, a top menu bar, a left tool bar, and a status bar at the bottom of the window (Figure 3.7). This layout is similar to many programs, including ITK-Snap, and should be familiar to most users. Since only

one tool can be applied to the Scene at any time, only one tool's options are displayed in the user interface at a time. By having the tool parameters shown in the main window rather than a pop-up window, all tools follow a similar layout and cannot obstruct the viewports. Many tools have interaction with the SceneObjects themselves, thus making it more important that parameters and viewports are both easily visible and accessible at the same time.

The rendering viewports provide four views of the current Scene and allow users to select SceneObjects. The Viewport class defines a generic viewport using a vtkRenderer and vtkRenderWindow for visualizing the vtkProps from SceneObjects. Viewports have methods to add vtkProps, refresh the view, and reset the camera so that all objects are within the view. Sub-classes SliceViewport and PerspectiveViewport define the specific types of Viewports, which are set-up with a ViewportFactory and arranged in a resizeable ViewportGrid.

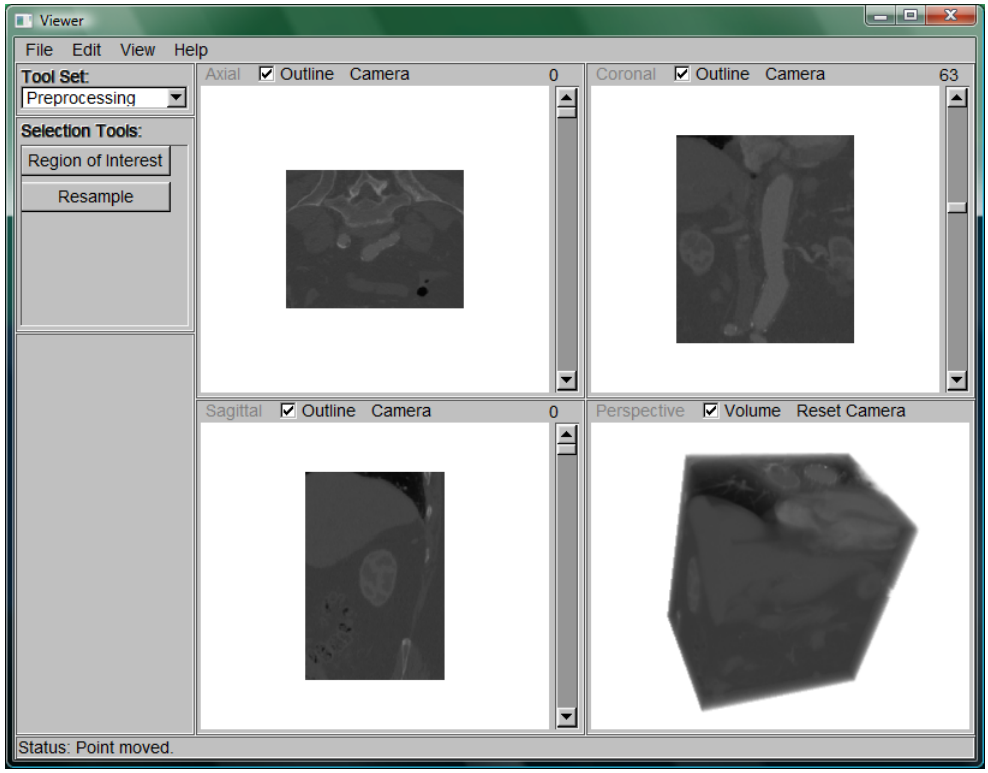


Figure 3.7: Graphical User Interface, with four viewports, menu bar at top, tool bar on the left, and status bar at the bottom. The viewports show Axial (upper-left), Coronal (upper-right), Sagittal (lower-left), and Perspective (lower-right) views of the dataset and objects.

Three SliceViewports are used to show slices in the orthogonal axial, coronal, and sagittal planes. These viewports have scrollbars that allow the user to choose the current viewable slice, which is indicated by the number above the scrollbar. The ability to pan and zoom in the viewport can be enabled through the viewport's camera menu. Enabling pan will allow the user to pan the view with the middle mouse button, but will disable any functionality with the middle mouse button that the current tool may have to ensure the user does not apply tool actions and pan at the same time. Similarly, zooming is done with the right mouse button. An option also allows the outlines of slices into meshes in the scene to be shown or hidden to speed up navigation between slices.

Mouse movements and button presses in a SliceViewport are handled by the SliceInteractorStyle class. This interactor extends from vtkInteractorStyleImage and handles mouse button press and release for left, middle, and right mouse buttons, along with mouse movement. Each time a mouse event occurs, the current picker is used to find the vtkProp under the mouse cursor and retrieve the SceneObject, which is passed to the Model. The current tool can set the picker to select nothing, objects, points, or cells as needed. Tools can also define the picker tolerance, as a percentage of the Viewport. Different mouse events can use different pickers and tolerances if required. For example, by setting unwanted mouse events to pick nothing, time is not wasted picking an object that will not be used.

The fourth viewport, a PerspectiveViewport, shows a perspective view of the entire scene. The

volumetric image of the dataset is rendered in this viewport, but may be hidden to speed up movement and refresh time. Showing a volumetric rendering can be very useful to better visualise the structures in the dataset, especially in conjunction with the Contrast Tool. The volume can also be viewed at the same time as other objects, such as meshes of anatomy, allowing for the regions around the objects and anatomy without geometric representations to still be visualised. The left mouse button allows rotation, the middle mouse button allows panning, and the right mouse button allows zooming of the camera to view the Scene from different angles. A PerspectiveSliceInteractorStyle class provides this and picker functionality similar to the SliceInteractorStyle class.

Volumetric datasets and polygonal geometry is loaded and saved from the menu bar's File menu. The View class handles the creation and display of file choosers, passing the chosen files and file types to a callback in the Controller. The Edit menu provides the ability to navigate through the applied tools. The last tool can be undone, and the previous or next tools can be chosen to allow options to be changed and the tool reapplied. A Contrast Tool is also available. From the View menu, all mesh outlines can be shown or hidden, and all viewport cameras can be reset so that the entire Scene is in view. A Properties Window provides information about the image and scene, while an Options Window allows viewing options to be modified.

The Contrast Tool allows the contrast of the image to be easily changed for improved viewing of structures which may differ only slightly in intensity. The overall range of intensity values shown can be set as well as the correspondence between the image's scalar values and the displayed intensities by dragging the points of a line graph. The number of points on the line can be set to provide the needed resolution, and the contrast can also be automatically reset to cover the entire scalar range of the image. The contrast is applied in real-time so that the results can be previewed and changed interactively. The ContrastTool component of the Model handles this functionality, while the ContrastToolWindow of the View displays the parameters and line graph.

The Properties Window displays image information such as intensity range, image extent, number of vertices, and number of polygons. The PropertiesTool of the Model retrieves the information from the image, counts the number of each type of SceneObject in the current Scene, and calculates Mesh properties. Then it sends this information through the Controller to the PropertiesToolWindow class which displays the information, as seen in Figure 3.8.

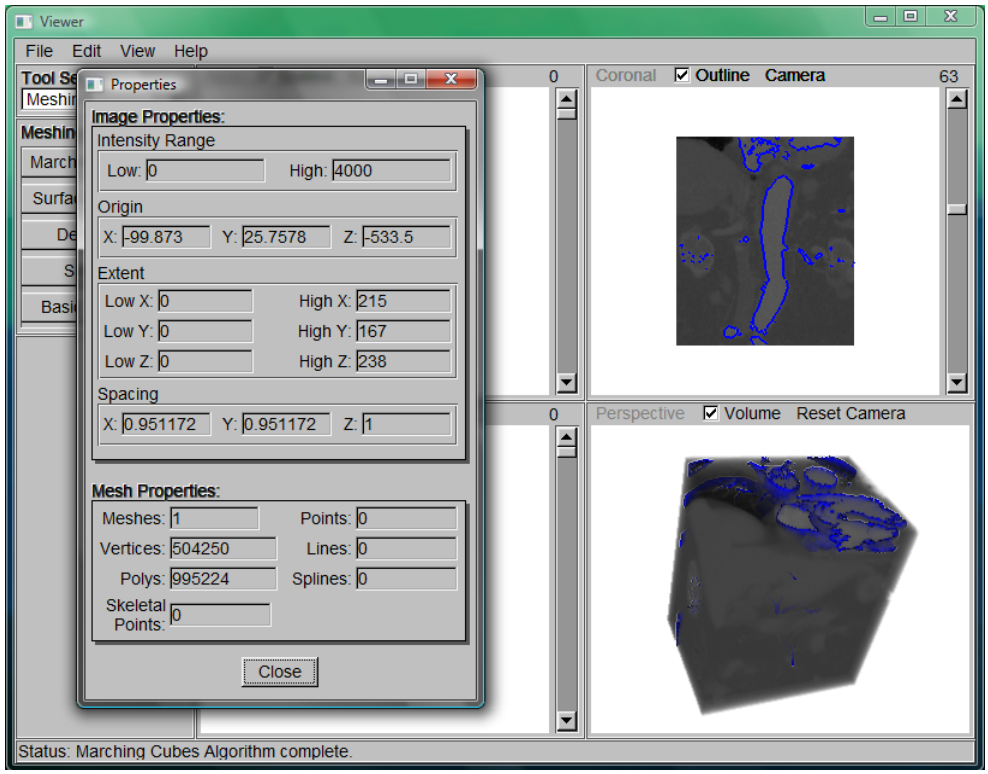


Figure 3.8: Properties Tool showing information about the dataset and the triangular mesh generated with the Marching Cubes algorithm.

From the Options Window, the viewport background colour can be changed, and the size of points and lines can be increased or decreased to improve visibility. Changing the background colour is especially useful for printing and can also help to distinguish the borders of the dataset. The OptionsTool class in the Model sets the point size and line thickness for the Scene, and it forwards the background colour to update the View. The OptionsToolWindow of the View displays the options, as shown in Figure 3.9, and sends user changes to the Model.

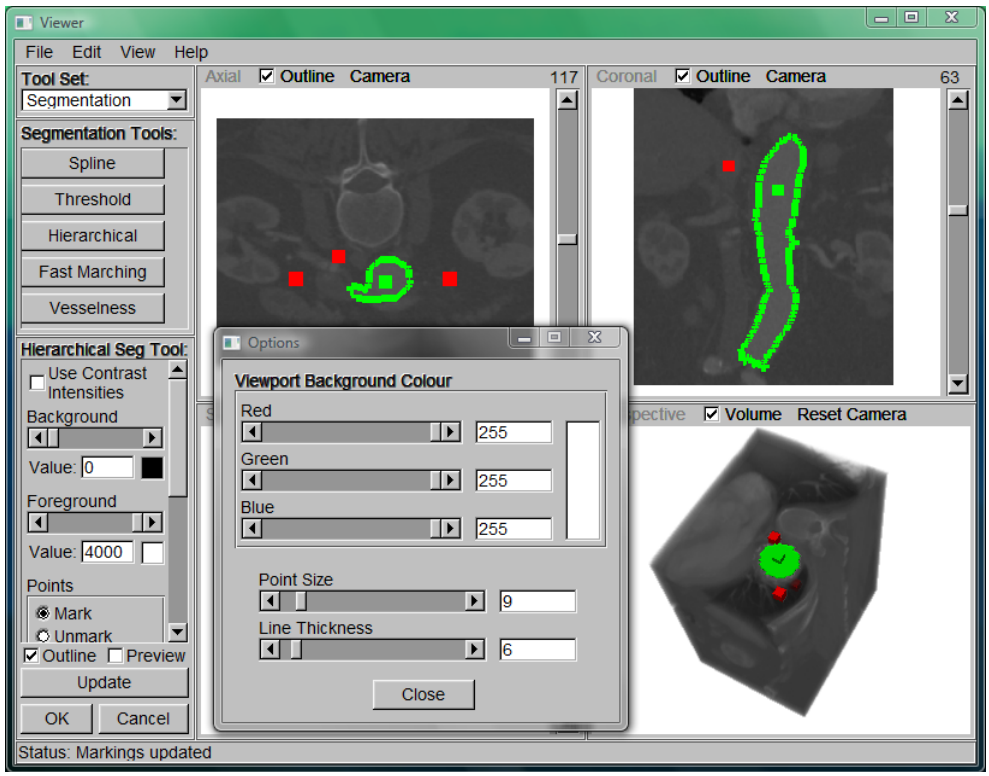


Figure 3.9: Options Tool used to change the colour of the background (here, white for printing) and the size of points and lines for using the Hierarchical or Editing Tools.

The tool bar on the left side of the main program window provides access to tools that the user can perform on the current scene. At the top, the tools are divided in a drop down list into Preprocessing, Editing, Segmentation, Skeletonisation, and Meshing tool sets. Selecting a tool set from the list displays the corresponding tools in the area below the list. Clicking on one of the tool buttons displays the parameters for that tool in the bottom area of the tool bar. At the same time as it is displayed in the user interface, the Tool class for that specific tool is initialized in the Model. Each parameter has a tool-tip which will be displayed when the user moves the mouse cursor over that parameter. These tool-tips provide a quick and easy way for a user to determine and be reminded of what the parameters control. As the user interacts with a tool, the status bar at the bottom of the main window is updated to give extra information and progress. After parameters are set to the desired values, the final algorithm can be applied via the OK button or cancelled with the Cancel button. The user may then continue with other tools if desired.

3.3 - Controller

The Controller component of the program serves as the link between the Model and View. Whenever the user changes a tool parameter, performs a tool action, presses or releases a mouse button, or moves the mouse, the information is sent from the View, through a set of methods in the Controller, and to the Model. And whenever data in the Model changes, the Model will update the View through other methods in the Controller. Thus, the Controller serves as a list of callback methods that are used whenever something changes and needs updating. When an object in the Scene is changed, the Viewports must be updated to redraw the Scene, so the Model must call on the Controller to update the View. When an object is added or removed from the Scene, the Viewports must be reset to retrieve the new vtkProps from the Scene. This is also achieved through a callback in the Controller. The Controller also provides

callbacks for file choosers. The Model and tools may call these methods to show file choosers for loading and saving datasets, geometry, or other files. The results from the file chooser are then returned to the Model to perform the file operations. Similar callbacks are used to show windows, set the current picker, and update the status bar.

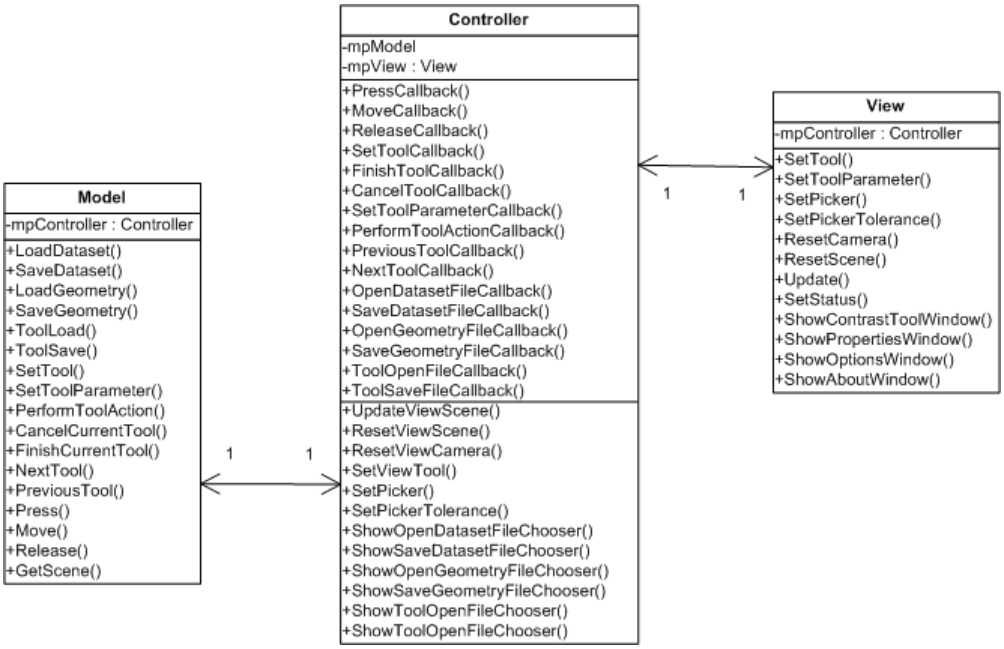


Figure 3.10: UML class diagram showing the relationships between the Model, View, and Controller. The lower set of Controller methods is used by the Model, while the upper set is used by the Model

To maintain flexibility, the Controller often uses integers to identify the parameters that are being updated. When setting the current tool, the integer identifier of the tool is used. Similarly, when setting parameter values, integer identifiers for the tool and for the parameter are used so that only one method is required for any tool and parameter. By using identifiers, new tools can be added without modification to the Controller. In order to add a new tool, only the MainWindow of the View needs to be modified, and only the new tool's class itself needs to be added to the Model. Similar identifiers are used for mouse events to identify which button was pressed or released. The Model and View classes have methods that correspond with the Controller's use of identifiers, so that the Model or View is responsible for the meaning of the identifiers and not the Controller. Thus, the Controller's methods remain very simple and do not handle any logic.

Because of the Controller, the Model and View do not communicate directly, allowing them to be easily changed separately. Furthermore, the Controller's methods are all very short, with one to three lines of code each, so that the Controller only serves as a routing agent and does not perform any actual work. The Controller could easily be reused even if the Model and View were to change significantly.

4 - Implementation

The program was implemented in C++, primarily using Microsoft Visual Studio 2005 for development and debugging. Several tools and algorithms were integrated and improved to provide the necessary functionality for segmentation, skeletonisation, and mesh generation from patient datasets. Because of the large amount of memory required to store and manipulate the volumetric data, memory optimization was a crucial part of the implementation. Steps were taken to reduce the memory requirements, and Valgrind [75] was used to check for memory leaks and errors. A Windows installer (Figure 4.1) was created with the Nullsoft Scriptable Install System (NSIS) [20], and CMake [19] was used for cross-platform compilation of the program. Documentation of the program code was automatically generated from comments using Doxygen [21].

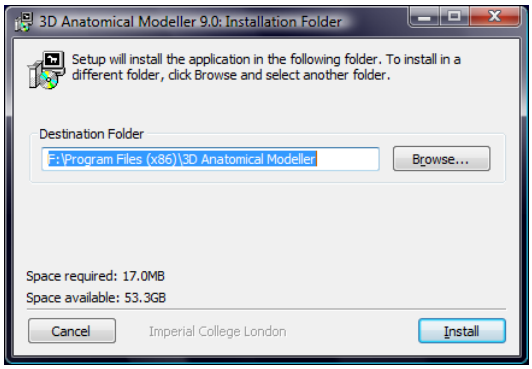


Figure 4.1: Windows installer created with NSIS.

As mentioned in the Design, the Dataset class provides access to both ITK and VTK image data, which are stored in itk::Image [76] and vtkImageData [77] objects. Conversion from the ITK to VTK format is provided by the itk::ImageToVTKImageFilter [76], and conversion from VTK to ITK is provided by the itk::VTKImageToImageFilter. The VTK format is used for visualization, and so must always be available; however, to reduce memory usage, the ITK format is allocated only when needed by an algorithm. The unsigned short type is used to store the image data rather than a float type to greatly reduce memory requirements. Some algorithms may require float types, but the itk::CastImageFilter provides the ability to cast the image from the unsigned short type to a float type and then back to the unsigned short type before displaying the output.

The DatasetReader and DatasetWriter load and save image files using VTK and ITK classes. The itk::ImageFileReader and itk::ImageFileWriter [76] are used to read and write most types of files, including Analyze (.hdr and .img), MetaImage (.mhd and .raw), and GIPL formats (.gipl). VTK structured points files are handled with the vtkStructuredPointsReader and vtkStructurePointsWriter [77], while DICOM (.dcm) files are read with the vtkDICOMImageReader. ITK's readers automatically convert to the desired unsigned short type, but VTK's DICOM reader provides only a float type image as output. This float image must first be shifted so that the values are all positive and then cast to unsigned short with a vtkImageShiftScale filter. The DICOM reader obtains the image orientation and origin incorrectly, so they must be corrected by using vtkImageFlip filters and accessing the true origin from the reader with the GetImagePatientPosition() method.

The colour transfer functions used to change the contrast and visibility of the inner structures of the dataset are implemented using a vtkColorTransferFunction [77], which maps the image values to intensities, and a vtkPiecewiseFunction, which maps the image values to opacities. Both functions are defined with pairs of scalar values and corresponding intensities or opacities. The points which the user sets in the Contrast Tool are used as the pairs for the vtkColorTransferFunction, while the opacity for each point is defined by the exponent of the intensity. More specifically, the opacity is defined as:

Equation 1: Opacity transfer function

$$opacity = \frac{(e^{intensity} - 1)}{(e^{1} - 1)},\ intensity \in [0,1],\ opacity \in [0,1]$$

This function produces a visualisation that is more transparent in darker areas and quickly becomes more opaque in lighter areas, which allows the user to see partially into the volume.

The Volume SceneObject provides the volumetric visualisation of the dataset using a vtkVolumeRayCastMapper associated with a vtkVolume actor. This volume mapper casts rays from the viewpoint and through the volume, adding contributions from each voxel as it is intersected. Thus, more transparent datasets will require more voxels to be traversed and increase the computation time. However, because of the exponential opacity transfer function used, the ray will typically not need to continue far into the data before becoming completely opaque. VTK offers an alternative mapper, the vtkVolumeTextureMapper2D, which renders the volume as a series of slices that are rendered using hardware. This mapper is faster for more transparent datasets because the hardware performs the composition of the layers rather than casting rays through voxels. However, this mapper does require time to flip the slice direction when the volume is rotated to view another side, and the speed-up due to hardware acceleration is not noticeable for highly opaque datasets and is actually slower for large volumes because of the number of slices that must be generated. Thus the ray cast mapper was preferred for this implementation over the texture mapper.

The Slice SceneObjects and its sub-classes generate texture-mapped planes of slices into the dataset. The plane geometry is stored as a vtkPolyData with a single quad cell and is supplied as the input for the Mesh superclass. The texture coordinates of the plane are generated with the vtkTranformTextureCoordinates and vtkTextureMapToPlane filters, and the vtkTexture of the slice is generated with the output from a vtkImageReslice filter. The AxialSlice, CoronalSlice, and SagittalSlice sub-classes are responsible for providing this filter with the position and direction in which the slice is generated and also to set the size of the plane to match the extent. Finally, the colour transfer function of the Dataset is applied to the texture with a vtkImageMapToColors filter.

Because algorithms allocate memory to store their output without directly modifying the input data, the amount of memory required increases rapidly as more algorithms are applied. This increase is especially relevant for algorithms performed on the dataset as multiple large volumetric images are stored in memory at the same time. Soon all memory will be completely filled, and the program will not be able to proceed; it is imperative that Tools release memory from previous algorithms when it is no longer required. Both VTK and ITK provide SetReleaseDataFlagOn() methods that cause an algorithm to release its data after it has already been used by the next algorithm in the pipeline [76][77]. For some algorithms, special care had to be taken to ensure the output was actually used before allowing them to release data; otherwise, the data would be lost and the lack of output would cause the next algorithm to fail. Tools must also properly release data in their destructor methods so that wasted memory does not build up over time.

Most operations on geometry make use of VTK classes, so it is only stored as vtkPolyData. However, when a CGAL or Tetgen algorithm is used, the geometry is saved to a temporary file via the GeometryWriter and loaded using the library. When the algorithm is complete, the output geometry is loaded with the GeometryReader. Since geometry is a more compact representation of structures than a dataset, it has a much smaller impact on memory usage, but some steps were still taken to reduce the memory requirements. Whenever possible, geometry is stored as triangle strip cells rather than individual triangles. A vtkPolyData object can be converted to triangle strips by the vtkStripper. If an algorithm requires triangle information, it can be converted using the vtkTriangleFilter and then stripped again afterwards. Figure 4.2 and 4.3 show the difference between triangles and triangle strips. Like datasets, geometry data is also released once the next algorithm is complete or it is no longer being used.

Figure 4.2: vtkPolyData made up of individual triangle cells.

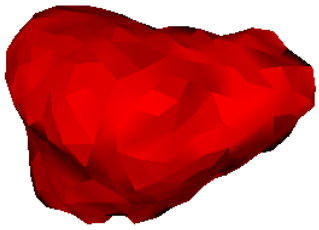


Figure 4.3: vtkPolyData made up of triangle strip cells. Each smooth region is a triangle strip.

Like the Dataset reader and writer, the GeometryReader and GeometryWriter classes do not perform file I/O themselves but use various VTK classes to provide the functionality required. The vtkPolyDataReader, vtkOBJReader, vtkSTLReader, and vtkVRMLImporter supplied by VTK [77] are used to read VTK Poly Data (.vtk), Wavefront Object (.wav and .obj), Stereo Lithography (.stl), and Virtual Reality Modelling Language (.vrml and .wrl) formats. Classes were added to include vtkMSHReader, vtkOFFReader, and vtkX3DReader to read Gmsh (.msh), Object File Format (.off) and X3D (.x3d) files. Another set of classes write these formats: vtkPolyDataWriter, vtkOBJExporter, vtkSTLWriter, vtkVRMLExporter, vtkMSHWriter, vtkOFFWriter, and vtkX3DWriter. While readers provide vtkPolyData as output, importers return a vtkRenderWindow from whose actors several vtkPolyData may be extracted. Similarly, writers take vtkPolyData as input, while exporters require a temporary render window to store the mesh actors.

Geometry is visualised with a Mesh SceneObject that creates a vtkActor by setting the vtkPolyData as input to a vtkPolyDataMapper [77]. Since different sub-classes of Mesh require slightly different visualisation in the SliceViewports, the slicing method can be set to either perform no slicing to show the entire object, a single 2D slice through the object, or a thicker 3D slice through the object ranging from the current slice to the next. When slicing is disabled, the original vtkPolyData is passed directly to the mapper for each view. To generate 2D slices, a vtkCutter filter is applied with a vtkPlane corresponding to the current slice, and the output from each slice provides the input for the mappers of the three slice views. The thick 3D slice is generated by using a vtkClipPolyData filter which removes all geometry on one side of a vtkPlane at the position of the next slice. Since the other side of the geometry is not visible because of the Slice object itself, it is not necessary to perform another clip, which would require more processing time. To reduce the amount of memory used while displaying geometry, the mappers are set with the ImmediateModeRenderingOn() method to render directly to the buffer rather than copy the data into OpenGL format first. For simple geometry this will not be noticeably slower, but when several complex meshes are in the Scene, the rendering may become faster because the preparation step is skipped.

The Viewport class displays the vtkProps from each SceneObject in the current Scene by adding them to a vtkRenderer. The renderer is assigned to a vtkRenderWindow which is provided to a vtkFlRenderWindowInteractor [77]. This class joins the VTK visualisation to the FLTK interface, and allows interaction with the displayed objects. To determine which object is selected by the user, the Viewport interactor styles, SliceInteractorStyle and PerspectiveInteractorStyle, use various VTK pickers. The available pickers are vtkPicker for complete objects, vtkPointPicker for points, vtkCellPicker for cell data, and vtkPropPicker to use hardware acceleration. Each of these pickers is supplied a ray from the mouse position through the scene and returns the first object intersected along with the point in 3D space. Because vtkPicker only uses the bounding box to determine which object was clicked on, it is not useful for several objects with overlapping bounding boxes. Therefore, the vtkCellPicker is typically used for picking objects as it will determine exactly where the object was picked. Although the vtkPropPicker's acceleration is useful when the Scene contains many objects, it often causes rendering problems with objects disappearing and sometimes exceptions because it must share OpenGL hardware functionality with visualization. To reduce the amount of time spent performing unnecessary pick operations when many objects are in the Scene, a Tool may choose to disable picking for certain mouse actions. For example, disabling the picker for mouse movements until actually needed will greatly speed up interaction; otherwise intersections tests will be performed each time the user moves the mouse over a Viewport.

The implementations of the Contrast Tool, Options Tool, and Properties Tool are straightforward, as they simply use methods in other classes to provide their functionality. The Contrast Tool uses Dataset methods to set the values of the colour transfer function according to the user input. The Options Tool passes its parameters to methods in the Scene to update the point and line size and a method in the Controller to set the Viewport background colour. The Properties Tool accesses the input Scene to gather its information and passes it to the Controller to update the PropertiesToolWindow. The remaining tools are divided into groups, each of which provides functionality to achieve a common goal or step in the process of generating models from patient datasets. A set of Preprocessing Tools performs simple operations on the dataset to remove extra information and prepare for other algorithms. Editing Tools

allow the user to manually modify the dataset and geometry to improve algorithm results. The Segmentation Tools extract anatomy from the image, and these segmentations provide input for Skeletonisation or Mesh Generation Tools.

4.1 - Preprocessing

The Preprocessing Tools allow the user to prepare the dataset and gather information for algorithms. Because patient datasets are typically large and contain more information than the user needs, some memory and computation time is wasted. The Region of Interest and Resampling Tools provide two methods of reducing the amount of data involved, thus improving the efficiency of algorithms applied to the dataset. Some algorithms require information about the size of structures involved, so the Measurement Tool allows the user to determine the distance between points in the image.

4.1.1 Data Reduction

Normally the user is only interested in a smaller portion of the dataset where certain anatomical features are found. The dataset outside of this region requires extra memory and also increases computation time for algorithms. Therefore, ignoring these regions can greatly reduce the time required for users to perform operations on the dataset and allow more operations to be applied before reaching the limits of the computer's memory. The Region of Interest Tool supplies this functionality, allowing the user to draw a box around the desired region and extract it from the dataset, as shown in Figure 4.4. The preview box, which denotes the bounds of the region to extract, is a Mesh object with input geometry provided by a vtkCubeSource. The user may specify the region by either entering the minimum and maximum X, Y, and Z values of the boundary or by left clicking and dragging the sides of the box in any of the SliceViewports. The cell picker is used to determine which side of the box is clicked on and dragged by the user so that the bounds can be updated accordingly. The output dataset can be previewed before acceptance by checking the preview box. The preview and final extracted dataset are generated with the vtkExtractVOI filter, and the extent of the output is also passed to the View so that the ranges of the slices allowed by the SliceViewport sliders are updated.

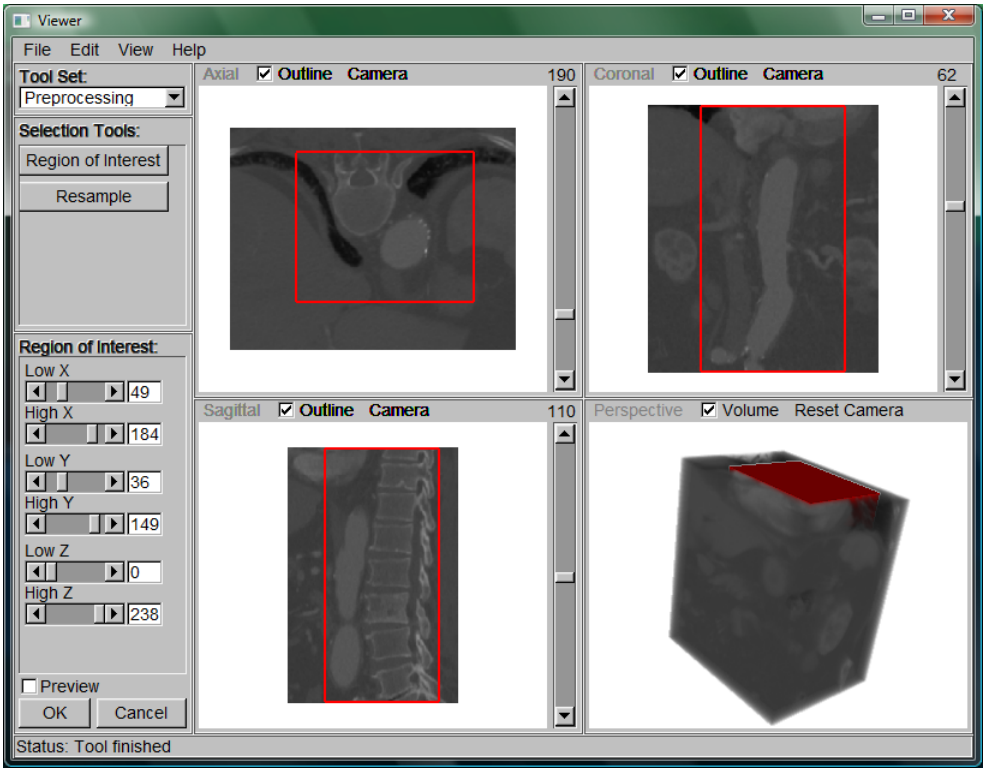


Figure 4.4: Using the Region of Interest Tool to select a portion of the dataset. In this case, the aorta (indicated by arrows) is extracted.

The dataset may also provide a higher resolution than required. Reducing the resolution of the

dataset will also reduce the memory requirements and computation time. The Resampling Tool allows the user to resample the dataset with different voxel spacings to change the resolution of the dataset. Not only can the resampling produce datasets with lower resolution by down-sampling, but it can also perform up-sampling if needed. The user can specify the output voxel spacing or extent in each dimension. If the user changes the voxel spacing, then the extent is updated so that none of the dataset is lost; similarly, if the user changes the extent, the voxel spacing is updated to produce that extent. The user can also choose between nearest neighbour interpolation provided by an itk::NearestNeighborInterpolateImageFunction [76] and linear interpolation provided by an itk::LinearInterpolateImageFunction. When the tool is finished, an itk::ResampleImageFilter is applied to the dataset and the output updated. Figures 4.5 and 4.6 show an example of down-sampling.

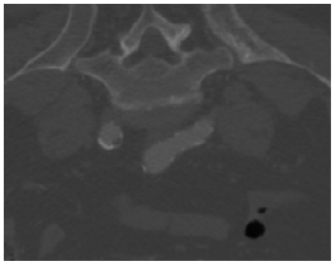


Figure 4.5: Original high resolution slice of 35,905 pixels before resampling.

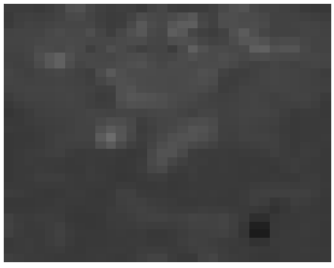


Figure 4.6: Dataset after down-sampling to 400 pixels.

4.1.2 Measurement

The user may require information about the size of structures in the dataset, especially the radius of blood vessels for use in the Hessian Vesselness Tool. The Measurement Tool allows the user to measure the distance between two points or along a path of more than two points. The user can insert the points to form the path with the left mouse button, move them by dragging with the middle mouse button, and delete them from the path with the right mouse button. As the user modifies the path, the updated point positions are displayed in the status bar and the length of the path is updated in the tool panel. The path is created by adding the Points to a Line object, which sums the distance along each segment to provide the length measurement. Figure 4.7 shows the the Measurement Tool being used to measure the diameter of the aorta in a dataset. This diameter can then be used with the Hessian Vesselness Tool to segment the entire aorta.

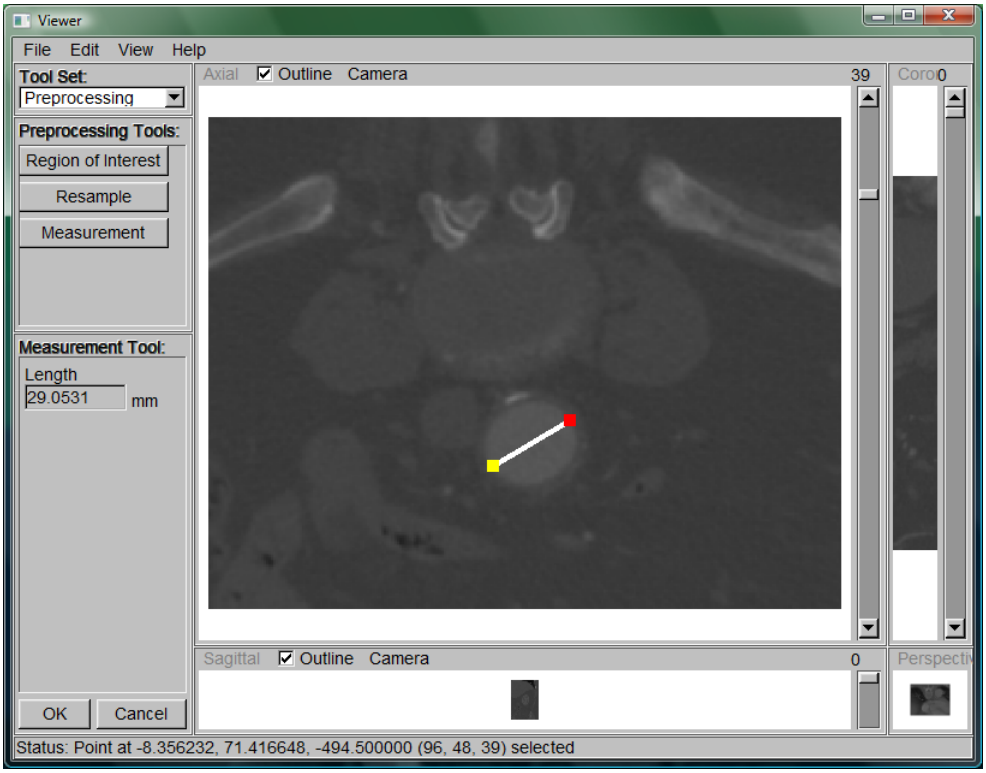


Figure 4.7: Using the Measurement Tool to measure the diameter of the aorta in a dataset. The user has clicked to insert a point on both sides of the aorta, and the length between those points is displayed in the tool panel on the left.

4.2 - Editing

Because some features may not be distinctly separated from their surroundings, a user may want to mark or enhance structures in the dataset to improve segmentation results. The user may also want to delete unwanted features completely so that they can not be segmented and interfere with the desired anatomy. Even after segmentation, the user may need to manually correct connectivity of regions or delete unnecessary parts. The Editing Tools perform these operations, providing the user with the ability to manually modify the dataset and geometry.

4.2.1 Voxelization

A crucial part of the editing functionality is provided by the vtkPolyVoxelFilter. This filter applies geometry to the dataset, drawing it into the image. One method of drawing geometry into a dataset is to iterate through each voxel in the dataset and determine if the geometry passes through it; however, this method is very slow, especially for the large datasets that are typical in medical imaging. Therefore, the method that vtkPolyVoxelFilter uses to apply a triangle mesh is based upon the vtkPolyDataToImageStencil [77] filter which masks an image using a mesh. The voxels that the surface of the mesh passes through are found by casting rays through the side of the dataset. To find the surface, rays are cast through each voxel on one side of the dataset through to the opposite side. Filling all voxels where intersections occur generates a voxelization of the surface. Figures 4.7 and 4.8 show the simple voxelization of a sphere. The inside of the mesh can also be determined by the intersections and filled accordingly. This method of voxelization does not require each voxel to be checked, so it is typically much faster, depending on the complexity of the mesh. A vtkOBBTree is used to generate a tight fitting bounding box around the mesh that is used to further speed up intersection tests for large meshes.

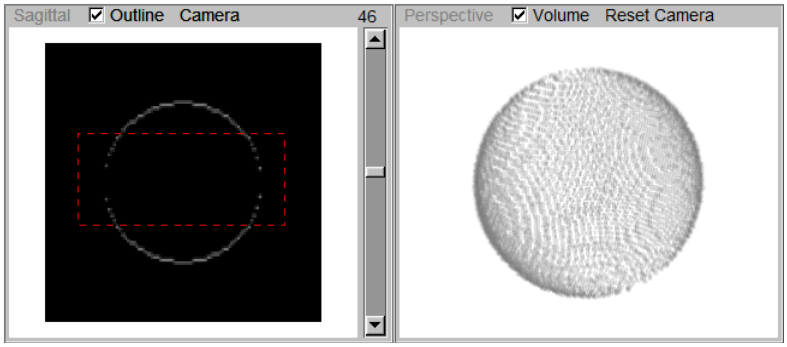


Figure 4.8: Voxelization of a sphere mesh by casting rays only perpendicular to the axial plane. Notice that the middle of the sphere (outlined in red) is not complete because no intersections occur.

Since some triangles may be parallel to rays cast in one direction, it may be necessary to cast the rays in other directions to draw the complete object. Otherwise, part of the object's outline might not be drawn, as highlighted in Figure 4.8. The vtkPolyVoxelFilter allows the user to choose one or more of the axial, coronal, and sagittal directions for casting rays so that this problem can be avoided. Figure 4.9 illustrates the voxelization of a sphere in all three directions. If the object is not a closed surface, then the filling will start at the intersection with the object and continue to the boundary of the dataset. Because points and lines will usually not be intersected by the rays, the filter handles their drawing separately from triangles. The filter applies each point in the geometry by filling the corresponding voxel, and it draws each line segment using basic line rasterization. A line is rasterized by starting at one end and using the slope of the line to draw the next voxel, continuing until the opposite end is reached. The scalars assigned to the points can be used to specify the output intensities of points and interpolated intensities of the line segments between the points. Alternatively, a foreground intensity can be specified for all types of cells. The background intensity can either be set to a constant value so that the objects are drawn on a new blank image, or the background can be copied from a dataset so that the output is drawn on top of it. Thus, the vtkPolyVoxelFilter can apply any input vtkPolyData's polygons, vertices, and lines to a volumetric image or generate a new voxelization image. This functionality is used by the Points, Lines, Splines, and Objects Tools to apply their SceneObject to the current dataset.

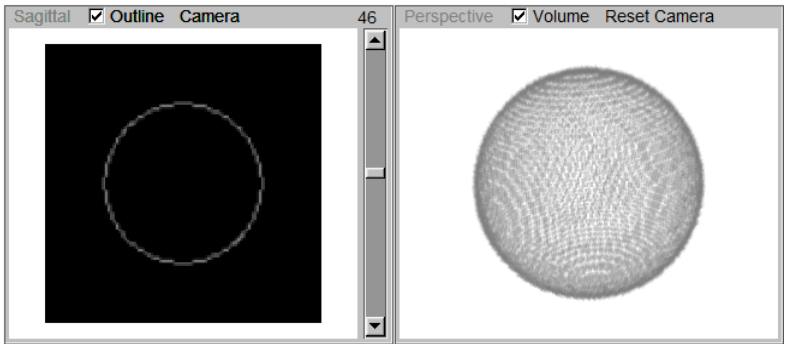


Figure 4.9: Voxelization of the sphere by casting rays in all three orthogonal directions produces a complete surface.

4.2.2 Points

Datasets or segmentations may have a few voxels with values that cause problems during segmentation or mesh generation. For example, a segmented blood vessel may have parts missing, causing gaps in the connectivity, as shown in Figure 4.10. Therefore, allowing users to manually fill these gaps can greatly improve the results and require less effort to correct than modifying a generated mesh. The Points Tool allows users to mark Points and draw them into the dataset. Points are inserted with the left mouse button and deleted with the right mouse button. An existing Point can be selected and dragged to a new position with the left mouse button. While selected, the colour of a Point can be changed by setting the red, green, and blue components. Figure 4.11 shows the inserted points coloured in red. To draw the Point into the dataset, the user sets the intensity and clicks the Apply button, as shown in Figure 4.12. The intensity can also be obtained from the voxel in the image where the Point is located, allowing

the user to both retrieve information about the intensities at certain locations and draw over already applied points, returning to the original dataset's values. Points can also be automatically drawn into the dataset immediately after they are added, allowing the user to quickly draw several points. The Points in a Scene can also be saved with a PointWriter and loaded from a file with a PointReader. The PointWriter takes a SceneObjectCollection of Point objects and writes them to a file when updated, while the PointReader takes the file path and returns the SceneObjectCollection of read Point objects. The file format consists of a header “POINTS” followed by the number of points, and then each following line contains the X, Y, and Z coordinates of the Point.

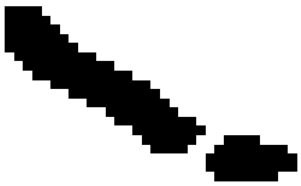


Figure 4.10: Zoomed-in view of vessel segments that the user wishes to connect with the Points Tool.

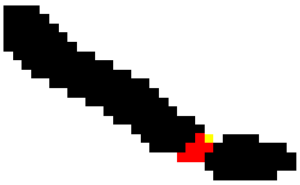


Figure 4.11: Points added by the user to be applied to the dataset.

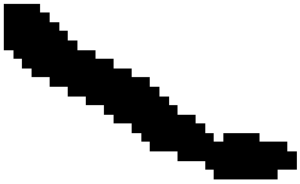


Figure 4.12: Points applied to the dataset to connect the two vessel segments.

As mentioned in the Design section, the Point SceneObject class is a sub-class of Mesh. Points can be either small cubes or spheres, and the vtkPolyData geometry for the Mesh is generated with either a vtkCubeSource [77] or a vtkSphereSource respectively. The size of a Point can also be set; the size is used as the length of each dimension of the cube or as the diameter of the sphere. Rather than slicing through the vtkPolyData for visualization in the SliceViewports, only the Points which are in the current slice are displayed. Thus, the Point class overrides the method used to set the current slice so that it makes itself visible only if it is in the slice.

4.2.3 Lines and Splines

The user may also wish to mark or enhance the boundaries of specific anatomical structures in the dataset. For example, by drawing a line on the border of an object with a high contrast to the object, segmentation algorithms will more accurately find this boundary. The Lines Tool allows the user to draw these lines by marking a series of points in the dataset, as seen in Figure 4.13. Points are marked with the left mouse button, moved by dragging with the middle mouse button, and removed with the right mouse button. As the user modifies the control points, the length of the line is updated and displayed in the tool panel. The control points are not limited to a single slice, but may be placed in multiple slices to create a 3D line. The colour of the Line object can be set by specifying the red, green, and blue values. Once done modifying the Line, the user can finish the tool, leaving the final Line object in the Scene, or the Line can be applied to the dataset with the specified intensity value. Lines can be saved to and loaded from files using the LineWriter and LineReader classes. The LineWriter takes a SceneObjectCollection containing the Lines to be saved and saves them to the specified file when updated. The LineReader reads from the specified file when updated and returns the Lines in a SceneObjectCollection. The file format begins with a “LINES” header and the number of Lines in the file. The next lines contain the number of point in a Line followed by the x, y, and z coordinates for each point.

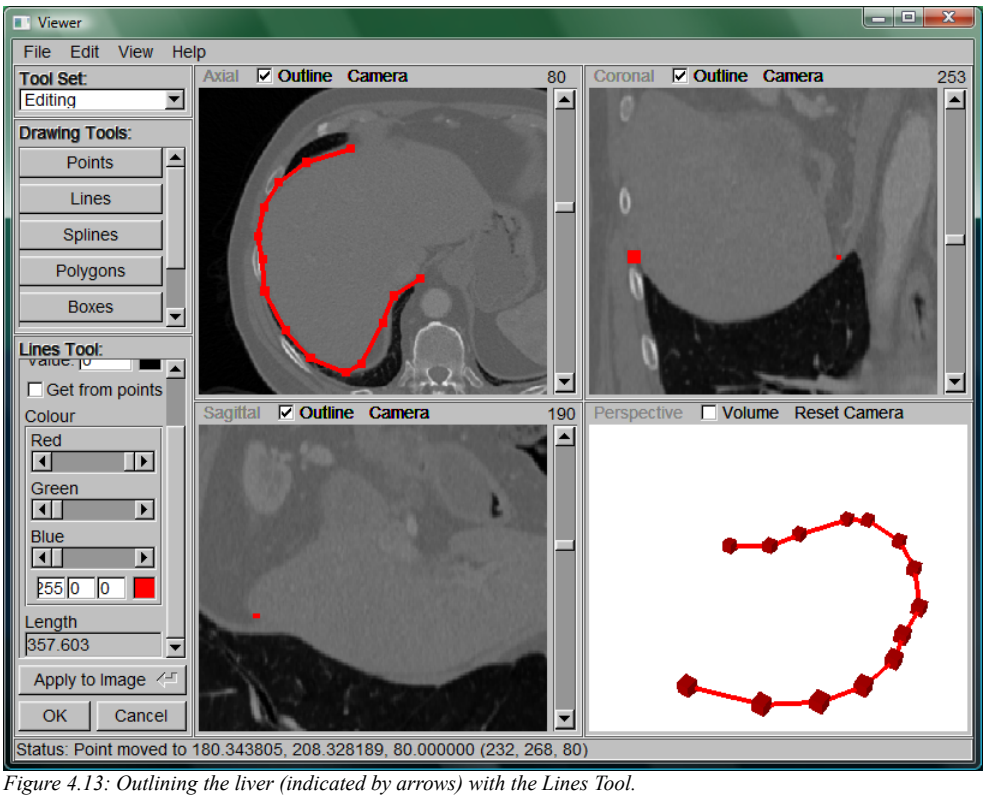


Figure 4.13: Outlining the liver (indicated by arrows) with the Lines Tool.

The Line SceneObject extends from the Mesh class, which provides most of its visualization functionality. A Line stores a SceneObjectCollection of the Points in the Line, allowing Points to be added, removed, and updated when changed. A method also calculates and returns the length of the Line by summing the distance from one Point to the next along its path. The Line's geometry is created by adding the points to a vtkPolyLine, which is then added to vtkPolyData. This vtkPolyData serves as the geometry input to the Mesh superclass. The Line class also sets the Mesh so that rather than generating a single slice intersection with the Line for visualisation in the SliceViewports, a range of the Line between the current slice and next slice is extracted and displayed. This range is much more useful than a single slice since the slice may only produce a single point on the line or could miss the line entirely. The Line also stores the intensities assigned to its Points in a vtkUnsignedShortArray so that these intensities can be applied and interpolated while drawing the Line into a dataset.

Most anatomy does not have straight boundaries, but smoothly curved surfaces. Thus, lines may not be ideal for marking these boundaries. The Splines Tool allows the user to draw a smooth two or three-dimensional spline by marking its control points, as shown in Figure 4.14. Like the Lines Tool, control points are marked with the left mouse button, moved by dragging with the middle mouse button, and removed with the right mouse button. As the user modifies the control points, the length of the spline is updated and displayed in the tool panel. The colour of the Spline object can be set by specifying the red, green, and blue values. The Spline can either be open or closed, and its quality can be controlled by length or by number of subdivisions. By choosing to subdivide by length, the user can set the length of each line segment that makes up the Spline. Alternatively, the user can choose to subdivide by number and specify the exact number of subdivided line segments. Once done modifying the Spline, the user can finish the tool, leaving the final Spline object in the Scene, or the Spline can be applied to the dataset with the specified intensity value. Splines can also be saved to and loaded from files using the SplineWriter and SplineReader classes. The file format for saving Splines is similar to that of Lines. The header here is “SPLINES”, and at the end of each line defining a Spline is either a 1 or 0 to designate whether it is closed or not.

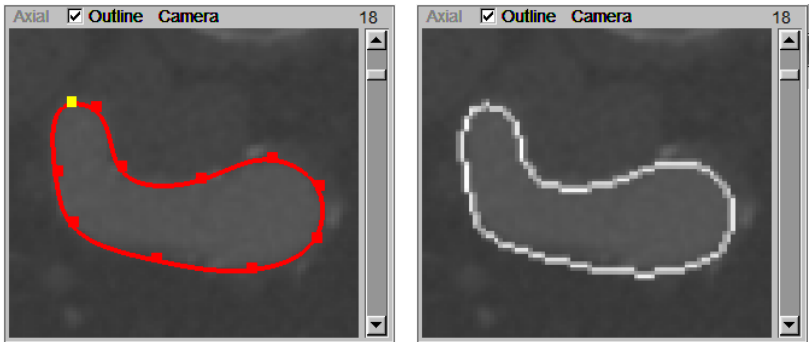


Figure 4.14: Spline applied to the dataset with the Splines Tool to enhance the boundary of the aorta. The white contour introduces a sharp contrast with the rest of the image, allowing the aorta to be located more easily.

The Spline SceneObject extends from the Mesh class and is similar to the Line class. A Spline also stores a SceneObjectCollection of its control points, allowing Points to be added, removed, and updated. The Spline's geometry is created by adding the points to a vtkPolyLine, which is added to vtkPolyData and filtered by a vtkOpenClosedSplineFilter with the specified subdivision properties. The vtkOpenClosedSplineFilter is a slight modification to vtkSplineFilter [77] so that closed splines are generated correctly: the subdivision loop had to be changed to connect the last point to the first point if the closed option is enabled. The output of the vtkOpenClosedSplineFilter serves as the geometry input to the Mesh superclass. The length of a Spline is calculated by summing the distances between points from the output of the spline filter. Splines store the intensities for its control points in a vtkUnsignedShortArray, which is also interpolated by the spline filter when the new points are being generated. These intensities allow the user to draw a Spline of varying brightness into the dataset.

4.2.4 Polygonal Meshes

After generating triangle meshes from the segmented dataset, there may be extra polygons from other, unwanted structures which were still partially present in the segmentation. Or, extra polygons may be generated covering a hole that the user wishes to keep; for example, using the Surface Mesher Tool to generate the surface of blood vessels will also generate a flat surface across the end of the vessel, where the dataset boundary occurs as seen in figure 4.15. These extra polygons can be quickly selected and deleted with the Polygons Tool. Left clicking on a Mesh will select the Mesh for modification. Left clicking again will select a specific polygon. The selected polygon can then be deleted with the Delete button. Selected polygons, or cells of the vtkPolyData, are shown by extracting the cells with a vtkSelectCellsFilter [78], passing the output to a vtkExtractEdges filter, which provides the geometry for another Mesh object. This Mesh shows a highlighted outline along the edges of the cells, denoting them as currently selected, as visible in Figure 4.16. Cells are deleted with the vtkRemoveCellsFilter, whose output serves as the geometry of the new Mesh, seen in Figure 4.17. The user interface also provides the option to choose between selecting individual triangles and triangle strips. A vtkTriangleFilter is used to convert the Mesh to individual triangles, while a vtkStripper generates the triangle strips for picking. The triangles allow the user to have more control over the parts that are deleted, while the triangle strips allow for quick deletion of a larger portion of the Mesh.

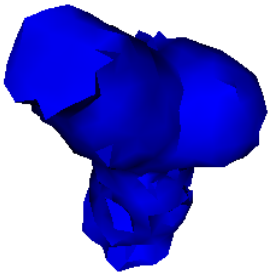


Figure 4.15: Aortic aneurysm mesh with unwanted polygons on the open end of the vessel

Figure 4.16: Using the Polygons Tool to select triangle strips for removal.

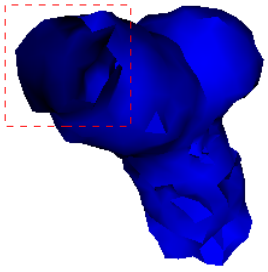


Figure 4.17: Aorta mesh after deleting triangle strips to create the opening (red).

Three tools provide basic objects for the user to add to the Scene: the Planes, Spheres, and Boxes Tools. The Planes Tool allows the user to define a plane by placing three points in the dataset with the left mouse button. The Spheres Tool allows the user to add a sphere by specifying its centre and radius, either using the input fields or dragging the centre Point and outline. The Boxes Tool allows the user to add a box to the Scene, as shown in Figure 4.18, by specifying the lower and upper bounds in the X, Y, and Z directions, either with input fields or by dragging the sides and centre Point. Each tool also allows the user to choose the colour of the object, and the Spheres and Boxes Tool both display a cross-hair made of three intersecting planes inside the object so that the centre is clearly visible. The output of these tools is a Mesh SceneObject with input vtkPolyData generated by a vtkPlaneSource, vtkSphereSource, or vtkCubeSource respectively. After generating the object, the user may wish to apply the object to the image with the Objects Tool.

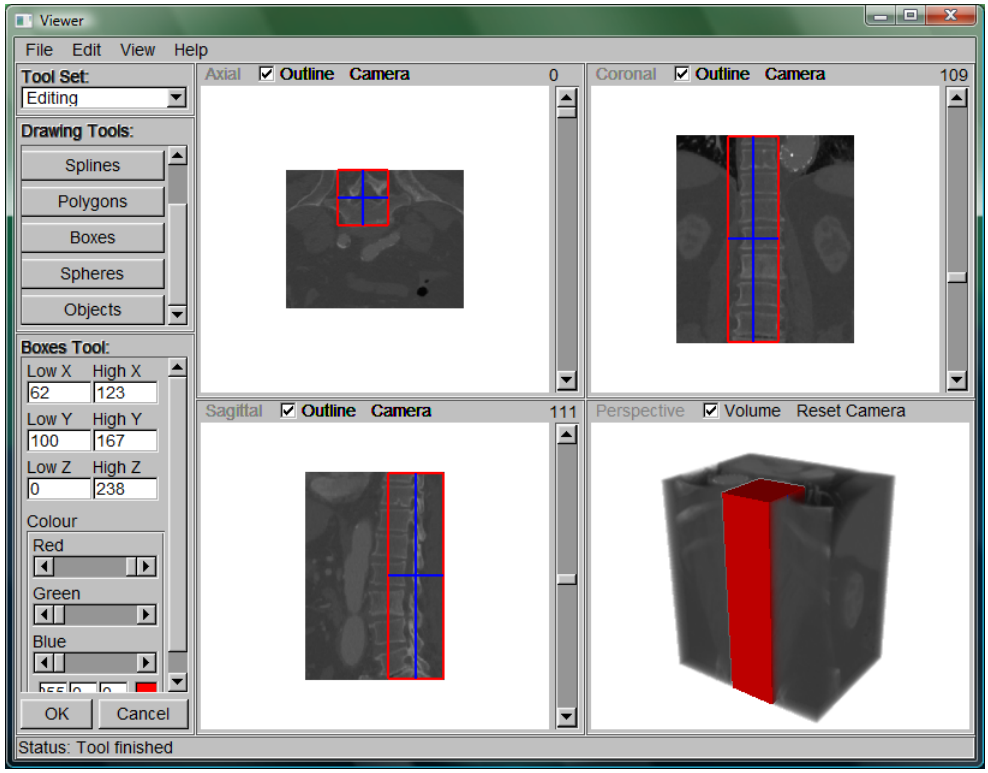


Figure 4.18: Using the Boxes Tool to draw a box around an undesired part of the dataset. In this example the portion contains the vertebrae.

The Objects Tool allows the user to change the colour of objects in the Scene, as well as apply the objects to the dataset. A Mesh, or any of its sub-classes, is selected by left-clicking on it, after which the red, green, and blue colour values can be set, or the selected object can be deleted from the Scene. To apply the object to the dataset, the user must choose the intensity used for drawing the object, whether or not the object should be filled (if not, only its contour will be drawn), and in what directions the vtkPolyVoxelFilter's rays should be cast to find the surface of the object, as illustrated by figure 4.19. The selected object's geometry can also be saved to a file using the GeometryWriter.

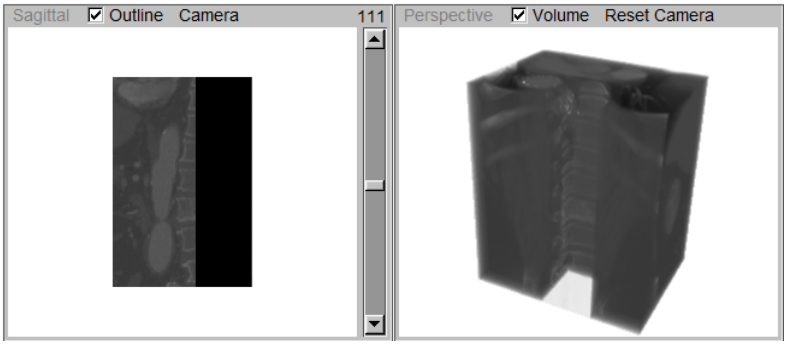


Figure 4.19: Result after applying the box to the dataset with the Objects Tool. In this case, an intensity of 0 (i.e. black) was applied to the dataset to remove a large portion of the vertebrae.

4.3 - Segmentation

Patient datasets contain many different structures and regions, but individual objects must be separated. An object is extracted from the image through a segmentation algorithm, after which it can be viewed and manipulated separately. This segmented image can then be used to generate an accurate mesh representation. Several segmentation methods are included in the present software, differing in the amount of user interaction required and focusing on specific types of structures.

4.3.1 Manual Segmentation

Because different anatomies in a dataset typically have different intensity ranges, applying a simple threshold to these values may remove unwanted regions of the image, as seen in Figure 4.20. The Threshold Tool makes use of an itk::ThresholdImageFilter [76] to clear voxels with values outside a user-specified range. The algorithm implemented by ITK is very simple and works by iterating through each voxel and setting it to 0 if its value falls outside the threshold range. A preview of the output dataset is also shown so that the values can be chosen interactively. An option also allows the mapped intensities that the user set with the Contrast Tool to be used rather than the actual values stored in the dataset. In this manner, the user can increase the contrast between regions to make the thresholds slightly easier to set. Thresholding is very fast, but datasets typically contain several structures with similar values that cannot be separated, as shown in Figure 4.20. Thus, this filter is more useful for removing some unwanted regions of the image before applying further segmentation algorithms. By reducing the information, other algorithms will require less computation and can be completed more quickly, giving better results with less user involvement.

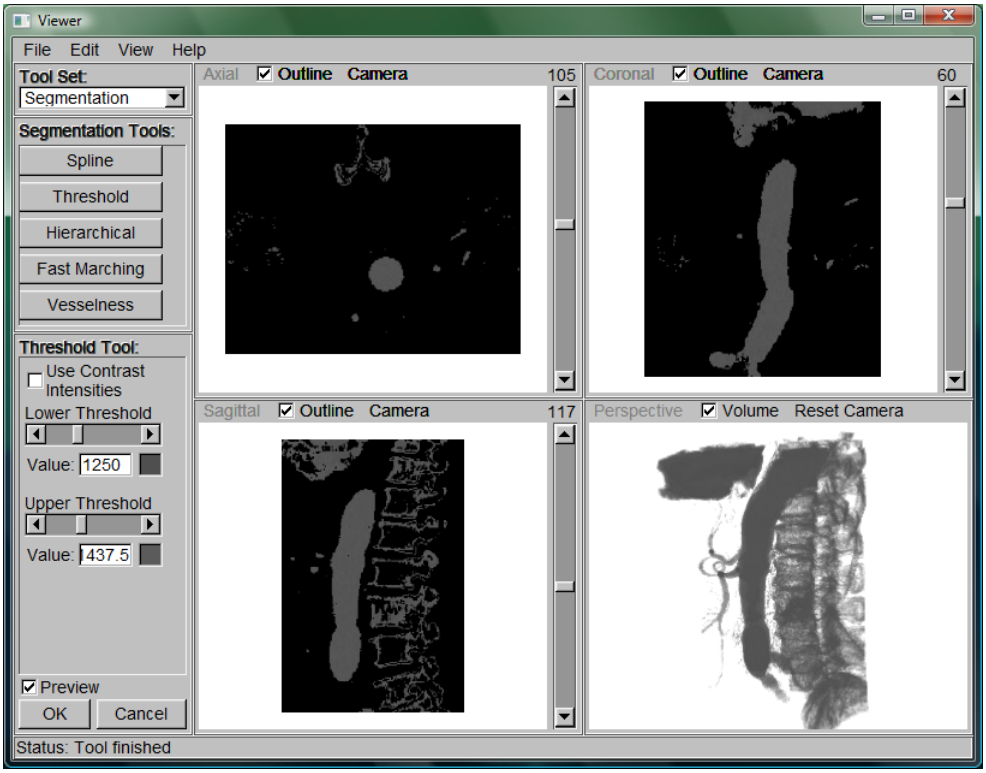


Figure 4.20: Applying the Threshold Tool to a dataset. Here, the thresholds were set to remove as much as possible while keeping the aorta.

The Splines Segmentation Tool allows a user to manually delineate structures, as shown in Figure 4.21. The functionality is very similar to the Splines Editing Tool, except that several splines can be easily applied in sequence. A Spline is again drawn by left clicking control points, which can then be moved with the middle mouse button and deleted with the right mouse button. Splines can be open or closed, and the user can set the amount of subdivisions of the spline. Options allow the user to fill the inside of the Spline with the foreground intensity and fill the outside with the background intensity. As when applying any type of object, at least one filling direction must be chosen. Once the user is done with the current Spline, it can be applied and then another Spline drawn.

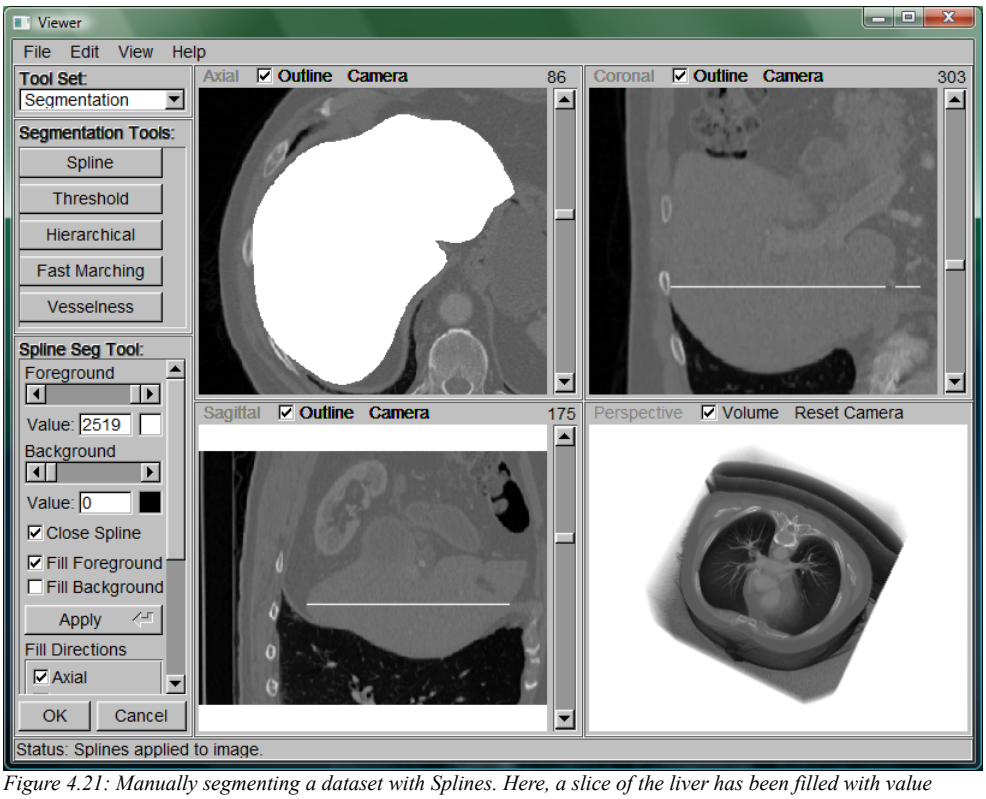


Figure 4.21: Manually segmenting a dataset with Splines. Here, a slice of the liver has been filled with value 2519, which is the maximum in the dataset, to appear white.

4.3.2 Fast Marching Segmentation

The Fast Marching Segmentation Tool allows both 2D and 3D segmentations of a dataset by marking seed points within the desired region, as seen in Figure 4.22. When working in 2D, the user may segment individual slices in either of the axial, coronal, and frontal planes. Each slice segmentation is stored in a volumetric output image so that the entire image can be completely segmented one slice at a time. The 3D mode, shown in Figure 4.23, applies the algorithm to the entire dataset rather than slices; however, the time required is longer and the user has less control over the results. When the segmentation is updated, the seeds are grown using a simple level-set algorithm to fill the desired region [79]. The Tool uses an itk::FastMarchingImageFilter [76], which requires a speed image to control the rate of the region's expansion and a list of seed points stored in a NodeContainer. The speed image is calculated as the sigmoid of the gradient image. Before generating the gradient, the image needs to be cast to a float type using an itk::CastImageFilter and smoothed with an itk::CurvatureAnisotropicDiffusionImageFilter to remove noise. The gradient of the resulting image is computed with an itk::GradientMagnitudeRecursiveGaussianImageFilter. The computation of the gradient is the longest part of this algorithm, but it must be done only once, after which 2D segmentation is performed at interactive speeds. Therefore, an option is also included to save a gradient image and load previously generated gradients from a file using ITK writers and readers.

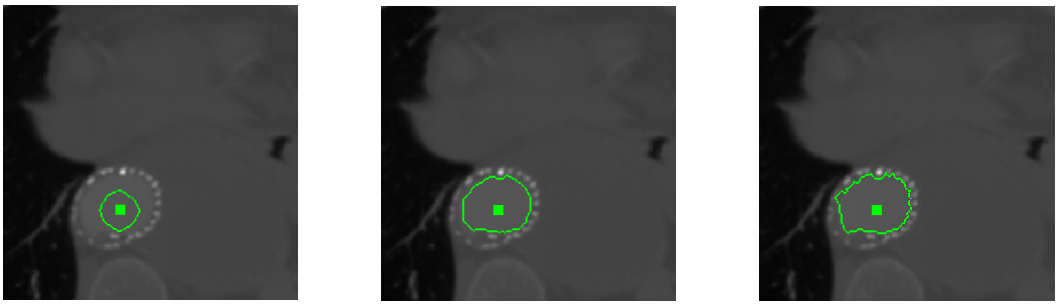


Figure 4.22: Expansion of the curve segmenting the aorta using the Fast Marching Tool. The first image shows a time threshold of 10, the second 20, and the third 40.

The sigmoid speed image is generated with an itk::SigmoidImageFilter [76] so that the level-set

curve will expand faster in areas with lower gradients and slow down at the higher gradients on boundaries of the object. The minimum gradient of the object's boundary along with the average interior gradient are used to set the sigmoid parameters. The speed image should ideally have a value of 1 within the object where the contour needs to continue expanding and a value of 0 at the boundaries where the contour should stop. After the speed image has been computed, the Fast Marching algorithm can compute the level set. Each value in the level set corresponds to the time that the growing curve passed through that point, so a segmentation at a specified time is produced by thresholding the results with an itk::ThresholdImageFilter. Specified by the user, the foreground and background values control the intensity of the object and the rest of the segmented dataset. As seeds are placed and the segmentation updated, a preview surface mesh is created using the vtkMarchingCubes algorithm, visible in Figure 4.23. The mapped output from the Contrast Tool can be used rather than the original dataset so that the user can contrast the object more with its surrounding regions, increasing the gradient and improving the segmentation results. Because the level-set used for Fast Marching segmentation only stores the time that the curve passes through each point, the contour can only grow or shrink and not pass through the same point more than once as can active contour algorithms. It does not work well with smooth boundaries, since the gradient will be too low to stop the curve from growing beyond the object. The gradient may also differ greatly along a boundary, causing the resulting segmentation to either fall short or spill outside the object in some areas.

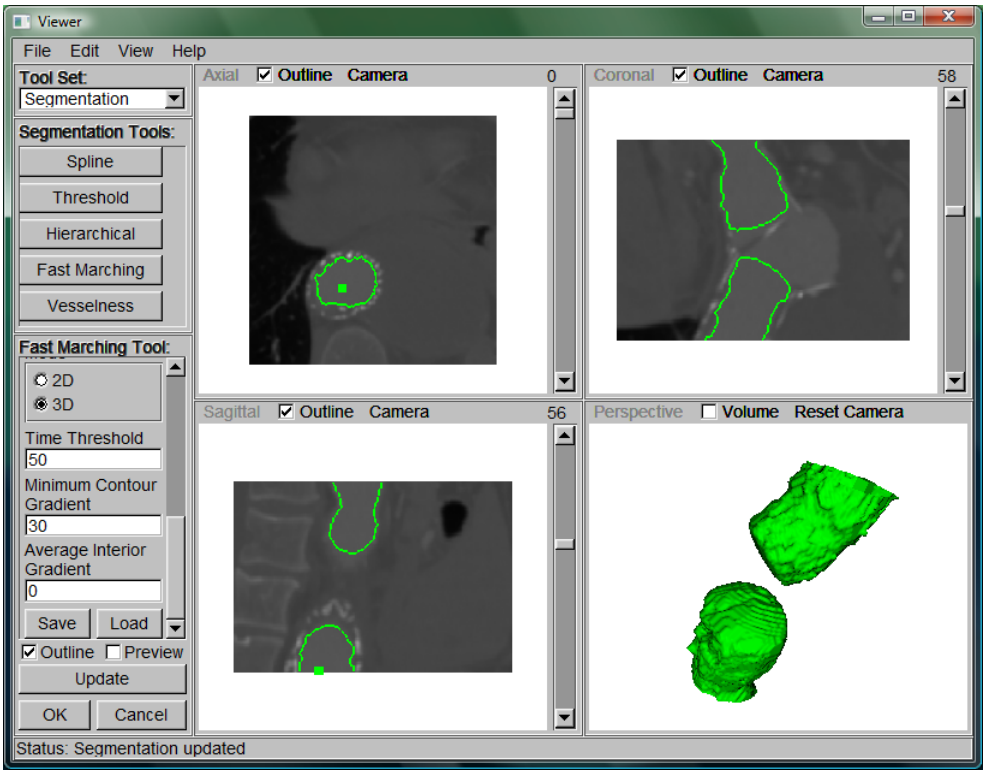


Figure 4.23: Applying the Fast Marching Segmentation. Here, a segmentation was performed using a seed at the top and bottom of the dataset. A third seed in the middle would fully segment the aortic aneurysm.

4.3.3 Hierarchical Segmentation

An alternative segmentation algorithm that is not dependent on the gradient of the dataset is provided by the Hierarchical Segmentation Tool. This algorithm allows the user to interactively mark points inside and outside the desired structure until the segmentation is satisfactory. Interior points are marked with the left mouse button and exterior points are marked with the right mouse button. Starting from these points, the algorithm iteratively merges adjacent regions based on their intensity and therefore separates the inside and outside of the structures. Options are included for deleting, clearing, saving, and loading marked points. Further controls allow the user to set the output foreground and background intensities, and the resulting 3D segmentation can then be previewed either via these intensities or an

outline mesh generated with a vtkMarchingCubes filter [77]. The output from the Contrast Tool can also be used as input to the segmentation algorithm to provide higher contrast between structures, allowing them to be more easily segmented.

Lewis D. Griffin's hierarchical segmentation algorithm [10] was adapted to provide both 2D and 3D segmentations. The 2D mode allows several slices to be segmented individually, in any of the three orthogonal views, as seen in Figure 4.24. The results of each slice segmentation are combined into a volumetric image so that objects can be segmented slice by slice. The full 3D hierarchical segmentation mode allows the entire image to be segmented at once. In either mode, the algorithm performs the segmentation by building a tree containing the relationships between regions of the image. Initially, each voxel is a separate region, and similar adjacent regions are iteratively merged based on the edge strength between them. In each iteration, adjacent regions are locally merged if the edge between them is weak. The edge between adjacent regions is weak if the difference in the mean intensity of the two regions is smaller than that of any other edge involving these regions. Eventually, the regions converge to a single region representing the interior of the object and another region representing the exterior. As similar adjacent regions are merged into parent regions, a hierarchy is formed.

Minimizing the difference in mean intensity between sibling regions ensures that parent regions are homogeneous [10]. Another necessary property that must hold for correct region separation is edge stability. The stability of an edge is defined by analysing its trajectory over Gaussian scale. Strong edges typically have a vertical trajectory, while the flatter trajectories of weak edges have less significance. The angle of trajectory is given by the following equation:

Equation 2: Trajectory angle

$$angle = \arctan\left(\frac{|\nabla I|}{|\nabla^2 I|}\right)$$

where I is the intensity,

$|\nabla I|$ is the absolute difference between the mean intensities of the two regions,

and $|\nabla^2 I|$ is the average mean of the Laplacians of the two regions.

Combining the edge stability with the homogeneity constraint provides a single edge measure:

Equation 3: Edge strength

$$edge\ strength = |\nabla I| \arctan\left(\frac{|\nabla I|}{|\nabla^2 I|}\right)$$

The intensity, gradient, and Laplacian of each region and its subregions is computed once and stored to reduce computation time.

The merging interactions proceed to form the hierarchy until every region contains only interior markings or only exterior markings [10]. The end result is a segmentation of the dataset based on the user's specification of points which fall inside or outside the desired object. To generate a segmentation, at least one interior and one exterior point must be marked; otherwise, the entire image forms a single region. The hierarchy for a single 2D slice can be constructed quickly and accurately with few markings, allowing the user to interactively improve the segmentation. However, the hierarchy for an entire 3D image requires a large amount of memory and processing time, yet typically provides poor results with large gaps in the segmentation even with a large number of markings.

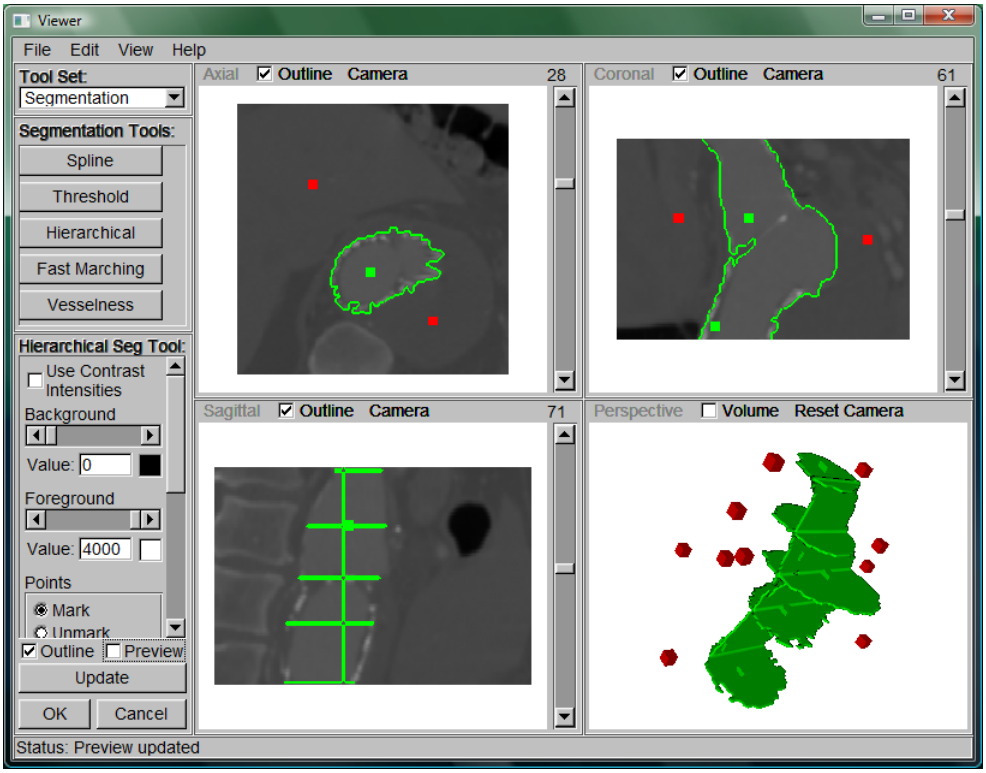


Figure 4.24: Applying Hierarchical Segmentation by placing interior points (green) and exterior points (red). The green outline provides a preview of the generated segmentations. Notice the 2D segmentations may be created for any slice and view.

To allow 3D segmentations to be made more quickly and accurately, a simple and efficient method was developed to extend the user's 2D slice segmentations into a full 3D segmentation, as in Figure 4.27. The extension is performed as a series of 2D segmentations using the user's previous segmentations as markings. A perpendicular slice will intersect the users' 2D segmentations to form lines of markings, shown in Figure 4.25. As long as the intersections include both interior and exterior markings, enough information is provided for a hierarchical segmentation in the perpendicular slice. Each perpendicular slice extends a portion of the user's 2D segmentations, so a series of slices across the extent of the dataset will produce a full 3D segmentation. If the intersections provide only exterior markings, then the entire perpendicular slice will be segmented as the exterior of the object. A similar situation occurs when the intersections only contain interior markings, which can occur when image does not include the entire object. However, these problems can be avoided by performing 2D segmentations in different directions, or by adding 2D segmentations on several slices to provide more markings.

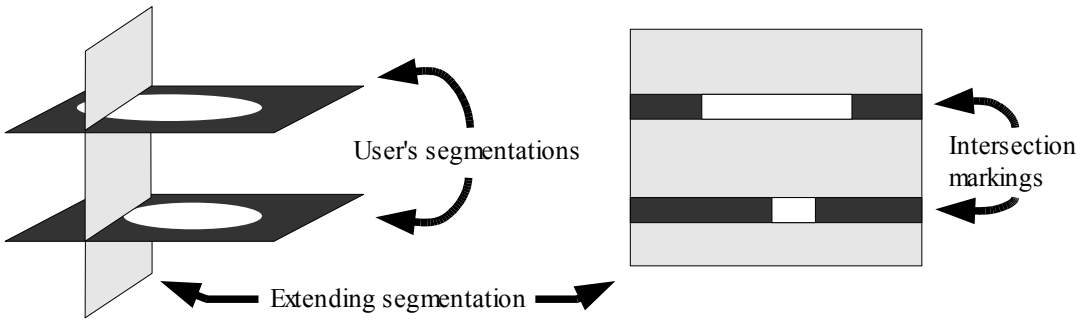


Figure 4.25: The intersection of the user's 2D segmentations produces markings in perpendicular slices. By performing a segmentation for each perpendicular slice, the 2D segmentations are extended into a complete 3D segmentation.

This algorithm only performs one 2D segmentation at a time and each resulting segmentation is stored in a single 3D image so that they require little extra memory. After a single 2D segmentation, the hierarchy is no longer needed and can be reset for the next segmentation. Whereas a 3D segmentation involves a large amount of information, separating the segmentation into a series of 2D segmentations

greatly simplifies the process to be faster and more accurate. The directions in which the extending segmentations are taken can be chosen through three check boxes; when more than one is selected, the areas in which the most extended segmentations overlap have the highest output segmentation intensity. These varying levels of intensity can be used during meshing to require a certain amount of overlap, usually corresponding to a better segmentation. This "2.5D" segmentation is automatically generated by clicking on the Update button after performing at least one 2D segmentation. After viewing the results, the user can improve the segmentation by performing more 2D segmentations and extending again to 3D.

Figure 4.26: Results from extending 2D Hierarchical segmentations to a full 3D segmentation. The green interior and red exterior markings are visible.

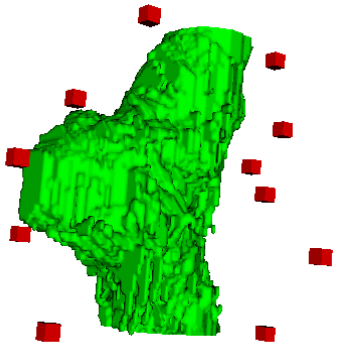


Figure 4.27: Preview surface of hierarchically segmented object (green) generated via the Marching Cubes algorithm.

The seeds from both the Fast Marching and Hierarchical Segmentations can be saved to or loaded from files. The file format is similar to that used to save points; however, the seeds in each view are saved separately so that the same 2D slice segmentations can be performed again after loading. Hierarchical segmentation also requires a flag to label each point as either an interior or exterior marking with a 1 or 0, respectively. A previously generated skeleton (see Section 4.4) can also be loaded as seeds to help segment blood vessels. Each skeletal point is considered to be an interior marking, though the user will need to mark exterior points for Hierarchical Segmentation. The user will still be able to remove any of the loaded seeds if they fall within undesired regions and add further seeds to refine the segmentation.

4.3.4 Hessian Vesselness Filter

As discussed in the Background, segmentation algorithms can produce better results if they take advantage of *a priori* knowledge about the shapes of structures. For example, blood vessels typically have a long and narrow tubular shape. Thus, if the algorithm searches specifically for these types of shapes during segmentation, undesired structures can be ignored. A 3D line filter using the Hessian matrix provides a method of filtering an image based on how similar structures are to tubular vessels of a specified radius [59]. The itk::Hessian3DToVesselnessMeasureImageFilter [76] calculates a vesselness measurement for each voxel in the dataset, determining how likely that voxel is part of a vessel. Figure 4.28 shows the filter applied to a dataset containing an aorta.

The Hessian matrix of a dataset is computed with an itk::HessianRecursiveGaussianImageFilter, which performs the convolution of the image with the second and cross derivatives of a Gaussian [76]. Each Hessian matrix describes the second order structure of intensity variation around a voxel, and its three eigenvectors and eigenvalues provide the direction and amount of variation. Therefore, simple spherical structures have three eigenvalues of similar value, because the shape is the same in all directions. Flat, plate-like structures have two eigenvalues near 0 for the plane of the plate and a large eigenvalue perpendicular to the plane, where the thinness of the plate causes higher variation. Tubular structures have two large eigenvalues for the cross section and an eigenvalue near 0 along its centreline, where there is little change. If a structure is brighter than its surrounding regions, then its eigenvalues will be negative, while darker structures have positive eigenvalues.

Vessels are typically brighter than the surrounding region, so the filter gives a higher vesselness measure to voxels with two large negative eigenvalues and the third near 0. The similarity to a vessel [59] is defined as:

Equation 4: Vesselness

$$vesselness = f(\lambda_1; \lambda_c) \times \lambda_c$$

where $\lambda_1 < \lambda_2 < \lambda_3$ and $\lambda_c = \min(-\lambda_2, -\lambda_3) = -\lambda_2$

The function $f(\lambda_1; \lambda_c)$ decreases as the smallest eigenvalue λ_1 deviates from 0 for more blob-like shapes. The eigenvalue λ_c is used to normalise this function: as λ_1 approaches λ_c, the shape becomes more like the blobs produced by noise. Therefore, the following function can be used to decrease the vesselness as the shape becomes more blob-like:

Equation 5: Vessel similarity function

$$f(\lambda_1; \lambda_c) = \exp\left(\frac{-\lambda_1}{2(\alpha_1 \lambda_c)^2}\right)$$

where α_1 controls the removal of noise and blob-like structures.

However, vessel structures may become fragmented due to signal loss, causing the eigenvalue in the direction of the line to deviate slightly from 0 in the positive direction. Thus, if we modify the function to give more weight to regions with positive λ_1 eigenvalues, these fragments can be made more continuous:

Equation 6: Vessel similarity function which makes fragmented vessel structures more continuous and reduces the effect of noise

$$f(\lambda_1; \lambda_c) = \begin{cases} \exp\left(\frac{-\lambda_1}{2(\alpha_1 \lambda_c)^2}\right), & if\ \lambda_1 \le 0, \lambda_c \neq 0 \\ \exp\left(\frac{-\lambda_1}{2(\alpha_2 \lambda_c)^2}\right), & if\ \lambda_1 > 0, \lambda_c \neq 0 \\ 0, & if\ \lambda_c = 0 \end{cases}$$

Smaller values of α_1 will cause more blob-like structures to be removed, further reducing the effect of noise. Larger values of α_2 will connect more vessel fragments together to form a continuous structure. The Hessian Vesselness Tool allows the user to change these values from the default ($\alpha_1 = 0.5$, $\alpha_2 = 2.0$) so that they can be tailored to the dataset. After applying the filter, vesselness values between 0 and 1 are returned for each voxel in the image. A vesselness near 1 indicates the voxel is most likely part of a vessel, while a value near 0 is most likely another structure or noise.

Figure 4.28: Hessian Vesselness with a radius of 10 and threshold of 0.5. In this case, only a portion of the aorta is segmented. The radii of the other parts of the aorta are different, so they are missing.

Figure 4.29: Hessian Vesselness with minimum radius 6, maximum radius 10, and step 4. During each step, values below 0.9 were removed with a threshold. Here, more of the aorta is segmented, but other structures are visible.

Blood vessels rarely have the same radius along the entire length, and branching vessels will often have different radii. Because ITK's implementation [76] only extracts tubular objects of a specified radius, a sequence of filter operations with different radii must be combined to extract the entire vessel structure, as seen in Figure 4.29. The Hessian Vesselness Tool allows the user to choose the minimum and maximum radius, along with the amount that the radius is increased each step of the sequence. The user

can use the Measurement Tool to find these radii. Before consolidating the results, an itk::ThresholdImageFilter removes regions with vesselness measures less than a user-specified threshold. Next, an itk::AddImageFilter combines the output from each iteration of the vesselness algorithm in the radius range previously specified. If the results were simply added together without thresholding, then regions with low vesselness (noise or isolated blobs) would add together and may even outweigh actual vessels. Because ITK's vesselness filter may return values greater than 1, another threshold is necessary to clamp the maximum value; otherwise, these values will push the final unsigned short value past the maximum that can be stored, causing regions of high vesselness to be lost. Any region with a vesselness greater than 1 is guaranteed to have a tubular structure, thus it provides no more information than a vesselness of 1. After the iterations are complete, the result is scaled with an itk::ShiftScaleImageFilter so that regions of higher vesselness will have values closer to the maximum short value, while regions of low vesselness will be near 0. The resulting image is then displayed as the segmentation of the dataset. This extension of Hessian analysis provides a fast method to segment blood vessels; however, some unwanted structures will still remain and some parts of vessels may be missing because of different radii. The results may not be sufficiently accurate, but they can be used as input to other segmentation algorithms for improvement.

4.4 - Skeletonisation

The tubular nature of blood vessels allows them to be well represented by a small set of points which follow the centre of the structure. The process of finding this representative set of points is called skeletonisation. By treating these points as spheres, they also represent the volume of the vessels, and the original vessels can be reconstructed from the union of these spheres. After determining the points, a centreline can be generated by linking the points together. This centreline stores connectivity information about the structures, and allows the user to obtain information such as length and radius of vessels.

4.4.1 Skeletal Points

The compact set of points is found with the Skeletonisation Tool using a distance transform method [11]. The input to this Tool should be a segmented dataset, with the foreground value of vessels larger than 0 and the background value of non-vessel regions equal to 0. This skeletonisation algorithm produces a set of skeletal points, each storing the distance to the nearest boundary of the vessel. When these distances are used as radii for spheres centred at each skeletal point, the union of the resulting spheres approximates the original segmented dataset. Thus, the dataset can be reconstructed from the compact set of skeletal points. Figure 4.30 shows a simple example of skeletonisation. To perform skeletonisation, the algorithm first copies the dataset into a temporary float array for analysis. Next, the distance from each foreground voxel to the nearest background voxel is computed by iteratively increasing the search area around a voxel until the background is found. Once the background is reached, the current search radius is stored at that point in the array, providing the distance from the voxel to the boundary of the vessel. This process continues by iterating through each voxel of the dataset, finding the distance to the background from each foreground voxel and storing it in the array. After the entire image has been processed, the array contains the distance from each point within the object to the surface of the object.

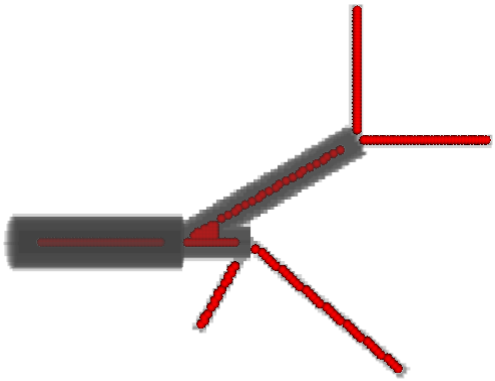


Figure 4.30: Skeletal Points found by the Skeletonisation Tool.

The skeletonisation algorithm proceeds to find a representative set of points using the distance information. By keeping only voxels whose distances are local maxima, the algorithm builds a compact collection of skeletal points. Therefore, a voxel remains in the collection only if its distance is larger than the distances of its 26 neighbouring voxels. A user-specified thinness parameter is used to further thin the skeleton, removing most of the undesired "hair" and "spike" artefacts caused by boundary noise. A test is used to remove points whose spheres likely fall within the union of neighbouring spheres. If the radius of a point minus the thinness parameter is less than the average neighbour radius, then several neighbours have large radii that could fully overlap the point's sphere, making it redundant. The user sets the thinness parameter to control how thin the resulting skeleton is: a value of 0 will produce a thick skeleton, while higher values produce thinner skeletons. Each skeletal point is stored as a SkeletalPoint object, which extends from the Point class. After skeletonisation, the resulting SkeletalPoints are added to the Scene for visualisation and for use by other Tools.

The original dataset containing the segmented vessels can be reconstructed from the SkeletalPoints with the Reconstruction Tool. Because the union of the points' spheres represents the full vessel structure, the algorithm simply iterates through each point and draws its sphere into the output image. A sphere is drawn by iterating through its bounding box and filling each voxel within its radius. The Tool allows the user to specify the foreground and background intensities, as well as the output image's spacing and extent. Before performing the reconstruction, the result can be previewed by displaying each sphere with the radius stored by its SkeletalPoint.

4.4.2 Centreline

Because a centreline provides an intuitive representation of vessel structure, a Centre Line Tool was added to generate them from the SkeletalPoints, shown in Figure 4.33. Vessels typically have a tree-like structure, so the centreline generation method is based on Prim's Minimum Spanning Tree algorithm [80]. The Skeletal points are treated as the nodes of a fully connected graph, with bidirectional edges connecting every pair of points. The weight assigned to each edge is the distance between the surfaces of the two spheres, which is found by the subtracting the sum of their radii from the distance between the two points. Therefore, if the spheres do not overlap, the weight will be positive (Figure 4.31), and if they do overlap, the weight will be negative (Figure 4.32). Spheres which are closer together or overlap more produce an edge with less weight, while spheres that are far apart will have a large weight. If the two spheres only touch the weight will be exactly 0. Because the centreline should only connect points which are close together and whose spheres overlap to form a continuous vessel, the edges with minimum weight should be chosen to form the tree. Edges with positive weights should be ignored because the spheres do not overlap and the vessel does not continue between those points.

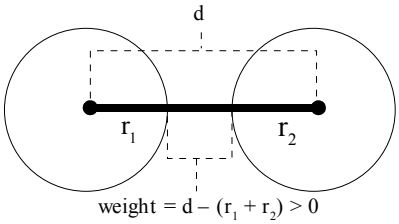


Figure 4.31: The weight of two spheres which do not overlap is the distance between their surfaces. Thus, the weight will be positive.

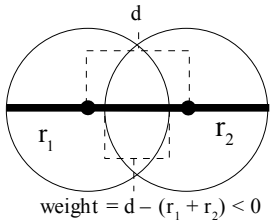


Figure 4.32: The weight of two spheres which overlap is the negative amount of overlap between the surfaces. Thus, the weight will be negative.

The tree starts with any skeletal point. Here, the first SkeletalPoint in the Scene's list of SceneObjects is chosen. The next point to be added to the tree is chosen by finding the edge with minimal weight connecting to a point already in the tree. If the user chooses not to enforce the connectedness of the centreline, edges between points whose spheres do not overlap will be discarded. Because the spheres may not always overlap between parts of a vessel where the radius changes rapidly (Figure 4.34), the centreline can be forced to connect all skeletal points (Figure 4.35). In this case, if no point's sphere is found to touch the sphere of a point already in the tree, the point closest to the tree will be added instead. This process continues, adding new points until all possible links have been added to the tree. The final minimum spanning tree provides a centreline representation of the vessel structure.

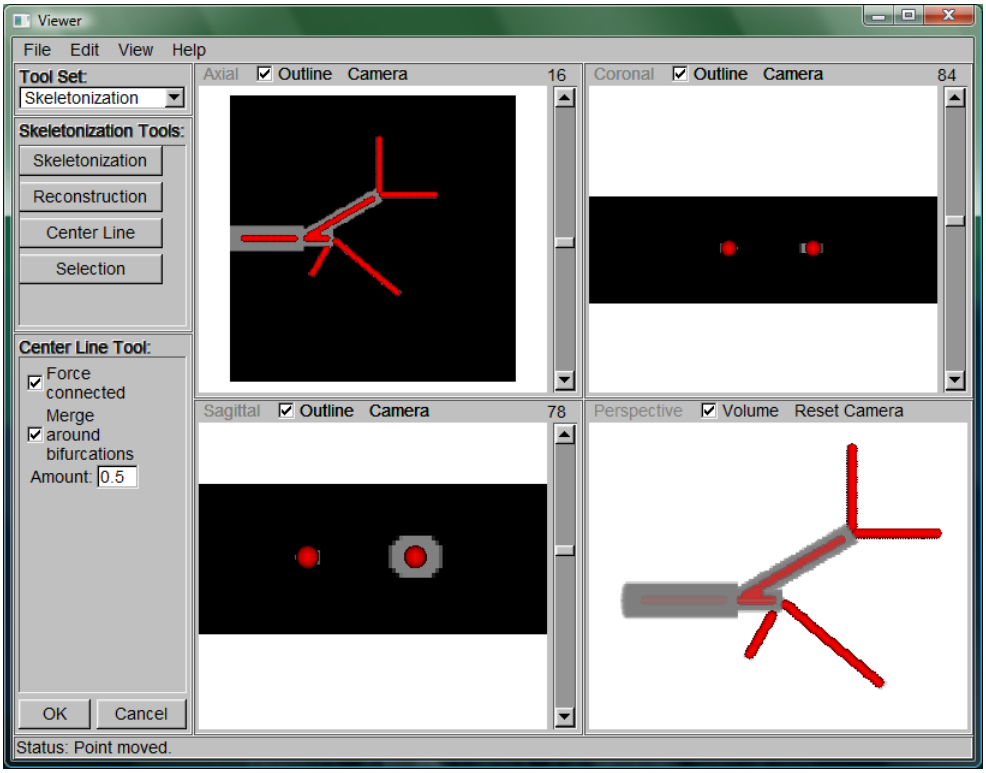


Figure 4.33: Setting the parameters for the Centre Line Tool on the left. The Skeletal Points generated by the Skeletonisation Tool are shown in the Viewports.

Skeletonisations typically have many redundant points around bifurcations, so another option allows these points to be merged into a single bifurcation point. If enabled, the algorithm searches for bifurcations in the tree and merges adjacent points, as shown by Figure 4.36. The bifurcations are found by recursively traversing the skeletal points and adding any point that has more than two incident edges. Then for each bifurcation, every other point is checked to determine whether or not its sphere overlaps the bifurcation's sphere. The spheres overlap if the distance between the points is less than the sum of their radii. Because, removing any overlapping points may delete many points, resulting in a poor quality centreline, the user can specify the amount of overlap required before removing a point. If the ratio of the distance between the point and the sum of their radii is less than the user-specified value, then the point is merged with the bifurcation point. The position of the merged point is set as the average of the two original point positions, weighted by their radii. The merged point's radius becomes the sum of the two

original radii and the distance between the points so that a large sphere is created, encompassing the original spheres. This ensures that no portion of the original structure is lost; however, this will cause the reconstruction to have spherical bulges around bifurcations. The edges of the centreline must also be updated while merging the points. The edges of the removed point are copied to the merged bifurcation point, excluding any edges to the bifurcation itself so that self-loops are prevented. The copied edges are also followed so that the end points' edges will no longer link to the deleted point but to the merged point. This process is repeated for each bifurcation point, until all possible merge operations have been completed to produce the final centreline.

Not only will merging remove redundant points around bifurcations, but it also tends to remove undesired branches caused by “hairs” in the skeleton. The skeletal points which make up these strands will usually have spheres overlapping the bifurcation's sphere, and so they will be merged until the entire branch no longer exists. Thus, the final tree provides an approximate centreline of the vessel structure. Each point of the centreline is stored as a SkeletalPoint object, which also stores its incident edges as a collection of SkeletalPoints. Because edges are bidirectional, only one point from each separate part of the centreline is needed to allow a full traversal along its edges to every other point in the part. This traversal can be performed by recursively following each edge from a point to other points. A collection of already visited points is used to avoid infinite recursion around a loop in the centreline. The CentreLine object stores a SceneObjectCollection of these root points and generates a visualisation of the line. The CenterLine class extends from Mesh, and its geometry is provided by generating vtkPolyLines along the edges between SkeletalPoints. The points and their edges are traversed recursively to construct these lines and extract branch information.

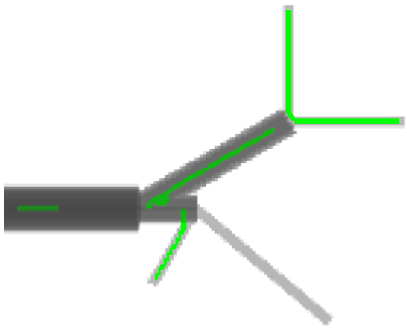


Figure 4.34: Centreline generated without forcing connectivity or merging points around bifurcations. Notice that the line is fragmented and one branch is entirely missing.

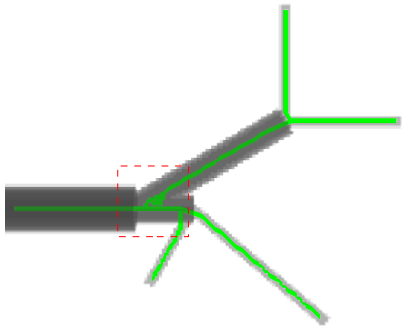


Figure 4.35: Centreline generated by forcing connectivity but not merging points around bifurcations. Notice the redundant line segments located near the bifurcation highlighted in red.

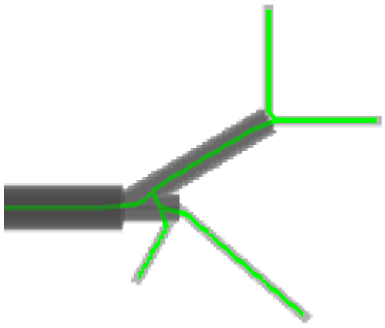


Figure 4.36: Centreline generated by forcing connectivity and merging points around bifurcations. Notice the redundant line segments are no longer present.

For storage and ease of access, it is convenient to represent the centreline as a set of branches. A branch is a segment of the centreline which can be represented by a single curve, without splitting off to other curves. Every edge between points is in exactly one branch, and the branches are connected by bifurcations. Therefore, the union of branches forms the entire centreline. The collection of branches is constructed while traversing the edges to create the geometry representation. When the traversal meets a bifurcation, a new branch is created as a Line object so that points can be added until the next bifurcation. The CentreLine class also handles conversions from cell identifiers, which are returned when the user selects the line. A cell identifier is mapped to the corresponding edge's start and end SkeletalPoints by traversing through the tree until the edge is reached. The branch identifier is obtained from this start and end point by searching each branch's Line for the one containing both points and returning its index in the branch collection. Tools and algorithms can access the SceneObjectCollection of branches, so that information about the structure of the Centreline can be obtained.

4.4.3 Skeletal Information

The Skeleton Selection Tool allows the user to select the SkeletalPoints and branches of the CenterLine to obtain useful information about the represented vessels. When a SkeletalPoint is selected, its radius is displayed to the user, providing the thickness of the vessel. When the user selects a branch, its points are highlighted and the length displayed (Figure 4.37). Thus, the length of a vessel can be obtained

by adding branch lengths. The user may add and move new SkeletalPoints with the left mouse button, and points can be removed with the right mouse button to improve the skeletonisation and centreline. The user can also save the centreline, using a selected point as the root. The file format starts with a "SKELETON" header and lists the branches, starting the traversal from the selected root. Each branch is saved as an ordered list of the points in the branch. The points are represented by numbers, starting with 0 from the root and increasing as the points are traversed. If the root is not the first point of a branch, the entire branch will still be saved in order, but the root point 0 will not be the first in that line of the file. The point numbers correspond to the point locations and radii stored in the second half of the file. SkeletalPoints may also be saved as seeds for the Fast Marching or Hierarchical Tools to perform segmentations of the vessels.

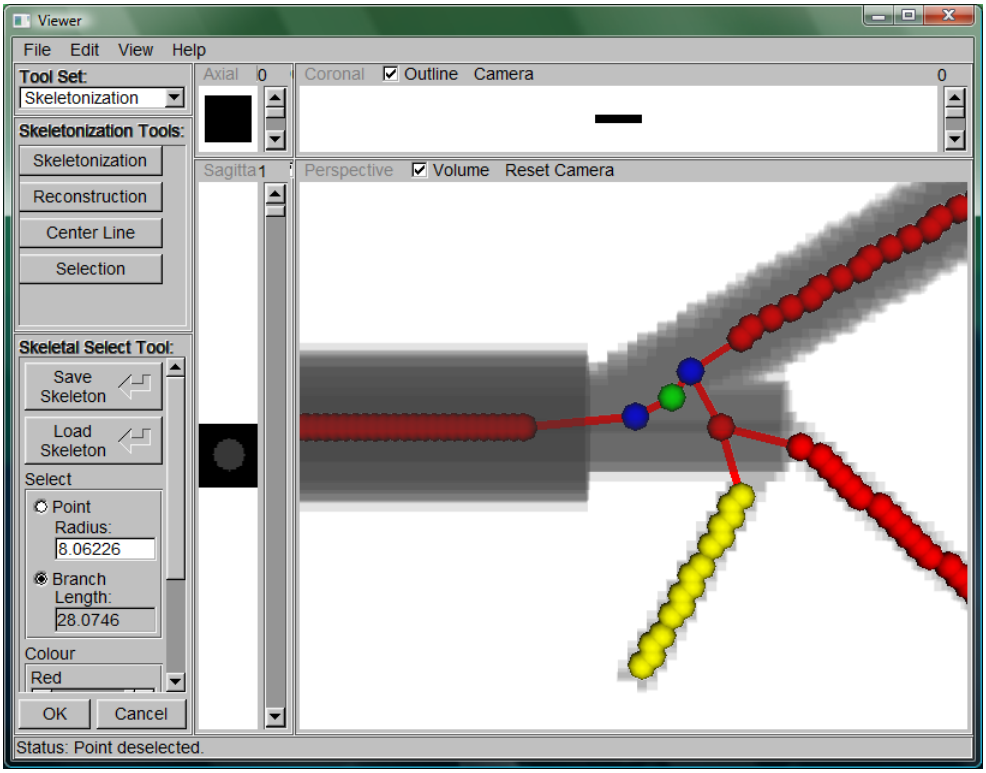


Figure 4.37: Selecting and measuring the length of a branch with the Skeletal Selection Tool. The branch's points have been highlighted in yellow, and the branch length is shown in the tool panel on the left (here 28.07 mm).

4.5 - Mesh Generation

Volumetric datasets require a large amount of memory for storage and long computation times for manipulation, consequently they are not directly suitable for real-time, interactive applications. A volumetric image consists mostly of redundant information since the values of voxels within the same structure typically vary little. The intensive mechanics calculations needed for simulations require another, much more compact representation of the anatomy to allow real-time interaction. Structures can be represented by their surfaces, and triangular meshes provide a compact representation that can be efficiently rendered in real time by graphics hardware. These meshes can be generated from the segmented datasets and then refined as needed. Some simulations also require an interior structure that fills the volume of the object; this is provided by a tetrahedral mesh. In the Mesh Generation tool set, two tools provide alternative surface meshing algorithms, another tool allows for the creation of the tetrahedral mesh from the surface, and further tools provide the ability to improve these meshes.

The Marching Cubes Algorithm [63] allows the user to quickly generate a surface from a volumetric dataset. The vtkMarchingCubes filter [77] is used to find an approximated isocontour for specified upper and lower isovalues. If the boundary voxels of a desired structure are close to the isovalue, the generated isocontour will approximate the surface of the structure. Although patient datasets will usually contain several structures with similar values leading to unwanted isocontours (Figure 4.39), segmented datasets will allow the desired surface to be produced by setting the isovalue between the

foreground and background values (Figure 4.40). To allow the user a visualisation of what the generated surface may look like while the isovalues are being chosen, an itk::ThresholdImageFilter is used to produce a preview of the dataset with the isovalues applied as upper and lower thresholds. This thresholding allows only the regions with values between the isovalues to be visible, and it is the boundaries of this region that will form the generated surface. Figure 4.38 shows the Marching Cubes Tool, with the parameters on the left and preview in the Viewports.

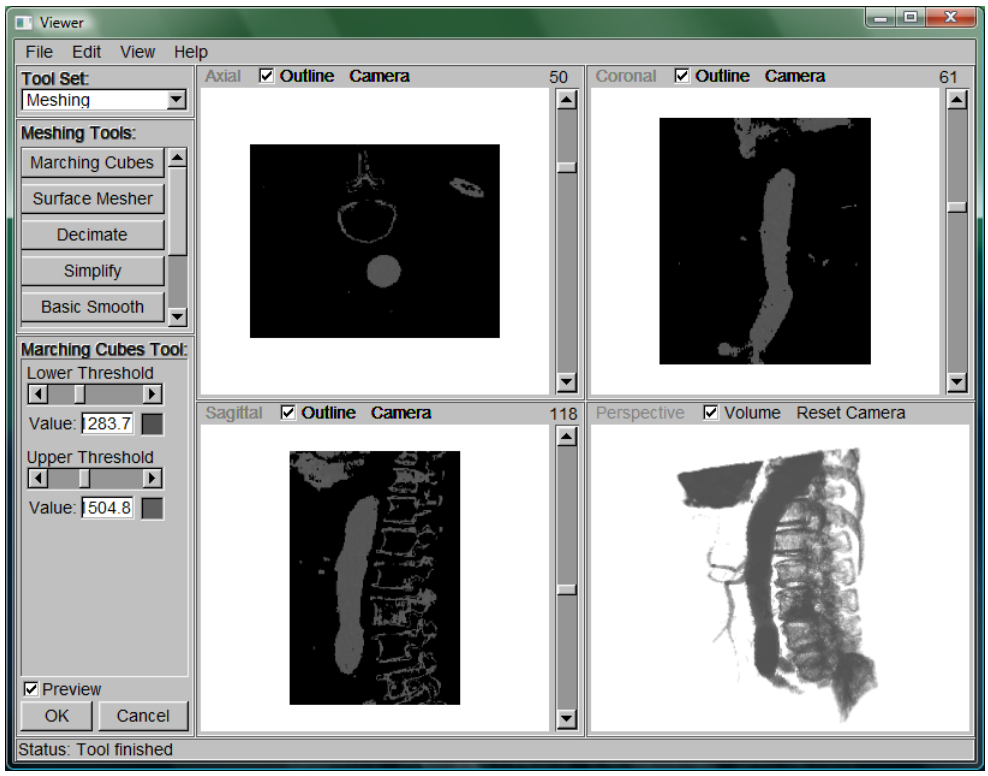


Figure 4.38: Using the Marching Cubes Tool to generate an isocontour from a patient dataset. The preview shows the dataset with the isovalues applied as thresholds so that the values can be set interactively. Notice that the vertebrae share some of the same values as the aorta, not allowing a mesh of the aorta alone to be obtained directly from the dataset.

VTK's implementation of the Marching Cubes Algorithm [63] iterates through each cube of the dataset and uses the states of the corner voxels as the index into the case table of 256 possible triangulations [77]. Each bit of the index represents whether a voxel has a value less than (bit value 0) or greater than or equal to (bit value 1) the isovalue. This eight-bit index into the case table yields the edges of the triangulation for the current cube. Each edge end point is interpolated between the corner voxels based on the difference of their values from the isovalue. This ensures that points will fall closer to the corner with a value more similar to the isovalue, constructing a more accurate isocontour. After placing the points of the triangles, the normal at each point is approximated by similarly interpolating the normals of the corner voxels; the normal of a voxel is defined by the gradient between the voxel and its neighbours. The algorithm also checks for and removes degenerated triangles, which have one or more points at the same location. Once each cube has been triangulated, the surface approximating the isocontour is complete. Because the algorithm creates a triangulation for each cube, the final surface will have many more triangles than necessary, as shown in Figure 4.41. Since the purpose of the geometry is to provide a compact and efficient representation of the data, the resolution of triangles must be reduced by using simplification tools.

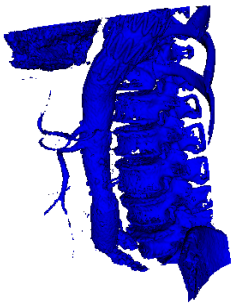


Figure 4.39: Mesh generated using the Marching Cubes Tool on an unmodified patient dataset.

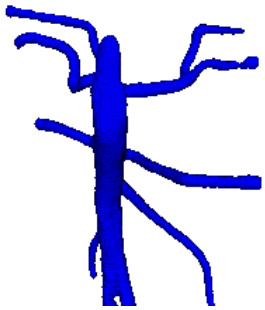


Figure 4.40: Mesh generated using the Marching Cubes Tool on a segmented dataset. This mesh is contains over 450K triangles.

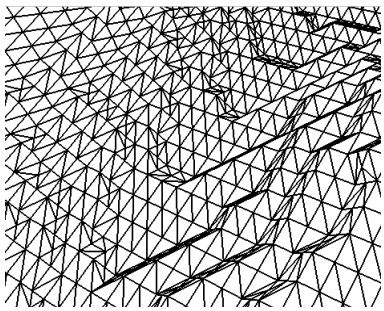


Figure 4.41: Close up of the redundant triangles generated by the Marching Cubes algorithm.

An alternative mesh generation algorithm that provides control over the size of triangles is supplied by the Surface Mesher Tool. This tool uses CGAL's surface meshing functionality accessed via the make_surface_mesh function [81] to perform a Restricted Delaunay Triangulation [70]. To set up the algorithm, the user must first define a sphere with its centre inside the segmented object and with the sphere surface surrounding the object, as shown in Figure 4.42. The Tool provides input boxes for the user to specify the centre and radius of the sphere, which is displayed to the user as a preview. The user can also left click and drag the centre of the sphere or the radius outline in any of the SliceViewports. A cross-hair made from three planes allows the user to easily find the centre of the sphere, which is crucial since it must fall within the object. The user must also set the intensity of the surface, and this value is again previewed using the itk::ThresholdImageFilter. Another set of parameters defines the properties of the mesh to be generated. The mesh angle defines the maximum angle (in degrees) between edges in the output mesh; if the object is very flat, this should be closer to 180, but if there are sharp corners, the value should be set closer to 0. The mesh radius sets the minimum allowed radius of a triangle's Delaunay sphere, while the mesh distance is the maximum distance allowed between the circumcentre of a triangle and the centre of a surface Delaunay sphere. The mesh radius must be no larger than the thickness of the object, otherwise the thinner areas will not be generated. For example, to generate a mesh for a vessel of 2mm diameter, a value less than 2 should be chosen for the mesh radius. The mesh distance provides some control over the distance between the generated surface and the actual boundary of the object in the image. The mesh topology can be specified as manifold, with or without boundaries, or non-manifold. Surfaces of anatomical structures are typically manifold, but the less strict options allow a surface to be generated even if the mesh parameters do not allow a manifold mesh. Using the non-manifold option is useful for determining the best parameters before generating the final manifold mesh, since there should always be some output that can be seen and improved.

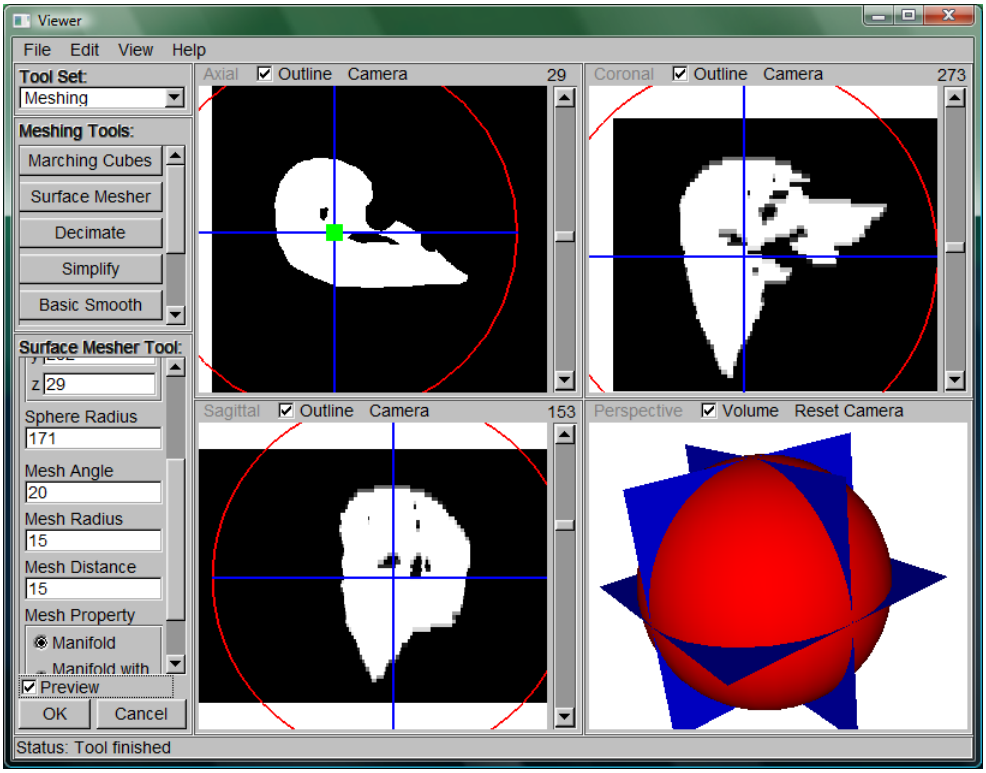


Figure 4.42: Using the Surface Mesher Tool to generate a mesh from a segmented liver dataset. The green point in the axial view shows the sphere centre, which must be inside the object. The sphere must surround the object completely.

CGAL's implementation of Restricted Delaunay Triangulation computes a set of sample points on the surface and generates a mesh that interpolates these points [81]. The surface is basically a three dimensional triangulation of the sampled points. Points are iteratively sampled as a refinement process until the specified size and shape criteria are satisfied. Because the sample points are always taken from the boundary of the object in the image, the surface will actually be generated slightly within the object. This causes the surface to be slightly smaller than necessary, though it could be expanded a small amount by scaling. Figure 4.43 shows a segmented dataset for meshing, and Figures 4.44 and 4.45 compare the results from the Marching Cubes algorithm with those from the Surface Mesher. It is easily apparent that the Surface Mesher provides more control and generates a better quality mesh. The surface is much smoother, without the individual slices being visible as with the Marching Cubes algorithm. The sampling of the triangles is also better, and control over the quality allows the generated meshes to meet the needs of simulations.

Figure 4.43: Segmented dataset of the liver before mesh generation.

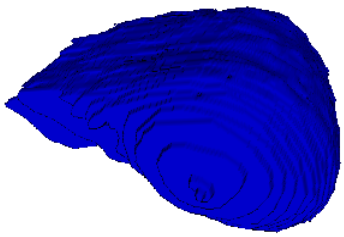


Figure 4.44: Mesh generated by the Marching Cubes Algorithm. Notice the visible rings for each slice of the dataset.

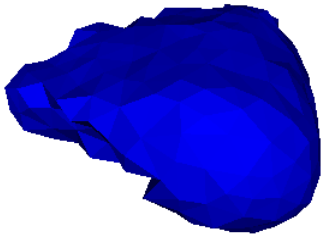


Figure 4.45: Mesh generated by the Surface Mesher Tool. Notice the quality of the triangles compared to Marching Cubes.

4.5.1 Simplification

Two simplification tools are provided to reduce the complexity of a surface mesh so that it can be used efficiently in simulations. The first simplification tool is the Decimate Tool, which uses vtkDecimatePro [77] to reduce the resolution of a triangle mesh. The user selects the Mesh to be decimated with the left mouse button, and sets the target amount of reduction, which is the percent of the

triangles to remove. This level of reduction will always be met if the user allows mesh splitting, deletion of boundary vertices, and modification of the topology. If the user does not want holes to be closed or opened during decimation, then the option to preserve topology should be enabled. Mesh splitting allows the mesh to be split along sharp edges so that it can be further simplified, and the split angle controls what is considered a sharp edge. A split degree also defines how vertices may be split if further simplification is necessary to reach the target reduction. Enabling boundary vertex deletion allows the algorithm to modify boundaries where the mesh no longer continues and has an open end. If the mesh needs to connect with another after decimation, then this option should not be enabled. The feature angle of edges that should not be modified can also be set to preserve sharp corners in the mesh.

VTK's decimation algorithm performs several passes in order to achieve a level of reduction closer to the target reduction [77]. In the first pass, every vertex is classified and added to a priority queue based on the error caused by deleting the vertex and retriangulating the hole. In most cases, this error is based on the distance from the vertex to the average plane of its boundary vertices. This metric allows vertices that vary more from their neighbours to remain while those that contribute less to the shape of the mesh are removed. Each vertex in the queue is processed, and if possible, the point is deleted and the new hole triangulated using edge collapse. Once the priority queue is empty, the remaining vertices are classified and the mesh is split into parts along sharp edges. The vertices are reinserted into the priority queue for a second pass. After the priority queue is empty again, the remaining vertices are split so that the third pass can remove any triangle as necessary. The algorithm stops as soon as the target reduction is reached, and if splitting is not enabled, then the algorithm will stop after the first pass.

Although splitting can ensure that the target reduction is reached, it may lead to holes or disconnected parts, so it is not always useful for simplifying anatomical meshes. Decimation also may not produce the quality meshes required for simulation as it is more concerned with the overall shape of the object rather than the quality of the triangles, as seen in Figure 4.48 compared to Figure 4.47. Therefore, very thin triangles may be produced, and the sizes of triangles may vary greatly, which is not ideal for use in simulations where calculations usually require more regular triangles.

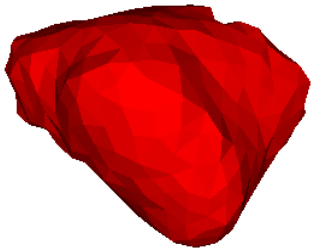


Figure 4.46: Original liver mesh with 660 triangles.

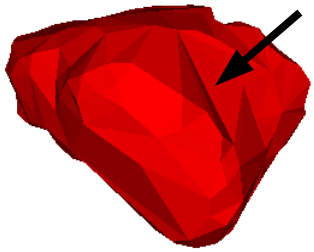


Figure 4.47: Decimated liver mesh with 380 triangles. Notice the long, narrow triangles.

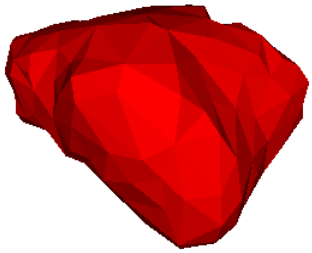


Figure 4.48: Liver mesh with 328 triangles after edge contraction. Notice the triangles are more regular than with decimation.

An alternative simplification algorithm is provided by the Edge Collapse Tool, which uses CGAL's "simplify" method [82]. The user selected the Mesh with the left mouse button and controls the simplification process with stop predicates. The count stop predicate causes the algorithm to terminate when the number of edges remaining drops below the specified count. Alternatively, the ratio stop predicate sets the amount of edges to keep after simplification as a percentage of the number of edges in the original mesh. The user may choose either stop predicate and apply the Tool; after the mesh is simplified, the resulting Mesh is added to the Scene. Because a filter does not exist to convert between vtkPolyData and a CGAL surface, the geometry of the selected mesh is written to a temporary OFF file with the GeometryWriter and then loaded by CGAL. Once the algorithm has completed, the result is saved to the temporary file and loaded again by the GeometryReader to serve as the input of the resulting Mesh.

CGAL's implementation of simplification actually uses a half-edge collapse method [82]. Rather than replacing an edge with a new vertex as does the edge collapse method [69], it removes the edge and pulls one vertex into the other [82]. For each edge removed, the adjacent triangles must also be removed, which involves deleting two additional edges. Every edge in the mesh is first ranked by the local

deviation which is defined by the Lindstrom-Turk cost strategy. This strategy does not compare the simplified surface to the original surface, but compares it only to the previous step in the simplification process. Thus, the original surface or history of changes need not be stored in memory, creating a "memoryless" simplification algorithm. The cost of collapsing an edge is based on the position of the replacement vertex, which is computed as the solution of three linearly-independent linear equality constraints. These constraints consider the shape of surface boundaries, the volume of the surface, and the shape of the triangle. The algorithm removes edges with lowest cost first, continuing until the desired number of edges have been removed. As edges are collapsed, the costs for edges containing the new vertex are recalculated so that the ordering can be updated. This algorithm will ensure that the triangles of the mesh are more regular than those generated by decimation because small edges are collapsed first. Although CGAL's algorithm is limited to manifold surfaces, anatomical models typically meet this requirement.

4.5.2 Smoothing

The mesh generated from a dataset will not always be as smooth as the anatomy that it represents. The segmentation of the structure may not be accurate enough to capture the smooth boundary or may have been affected by the resolution of the imaging modality or by noise. A Marching Cubes Algorithm will closely match the segmentation, producing the same sharp edges and small imperfections. A highly decimated surface may also have sharp corners which should not be present and will cause the mesh to be of poor quality, unsuitable for simulations. The Laplacian Smooth Tool allows the user to smooth out these sharp features or bumps in the mesh using the vtkSmoothPolyDataFilter [77]. The user selects the Mesh with the left mouse button and can control the smoothing with several parameters. The number of smoothing iterations controls how smooth the resulting mesh becomes. The amount of smoothing can also be controlled with a convergence value: if the maximum motion during an iteration is less than this value, then the algorithm has converged to its result and is terminated. The relaxation factor determines how far points are moved during each iteration. Large factors may cause the algorithm to become unstable, so small relaxation factors with a large number of iterations typically provide better results. By enabling feature edge smoothing and specifying a feature edge angle, the user can preserve sharp edges while still smoothing along the feature edge. The edge angle specifies the maximum angle of a feature edge to be smoothed with adjacent feature edges. If the mesh must connect on an open side with another mesh, then boundary smoothing can be disabled, allowing the meshes to line up properly.

VTK's smoothing filter modifies the mesh with Laplacian smoothing [71], which performs a simple and fast relaxation of the mesh. The algorithm first determines which vertices and cells are connected to each vertex in the mesh to construct a connectivity array [77]. In an iteration each vertex is moved towards the average of its adjacent vertices by the specified relaxation factor. If feature edge smoothing is enabled, then vertices are classified as simple, interior edge, or fixed depending on the number of incident feature edges: simple vertices have no feature edges, interior edge vertices have exactly two, and all others are fixed. Smoothing differs depending on classification: fixed vertices are not smoothed, simple vertices are smoothed normally, and interior edge vertices are only smoothed along the feature edges. After many iterations, the smooth result is obtained. Although the filter removes high frequency information like noise from the mesh, too much smoothing will quickly remove desired features even with feature edge smoothing enabled. Shrinkage will also occur, making this smoothing method a poor choice for tubular structures such as blood vessels. Figure 4.50 shows how severe this shrinkage can be.

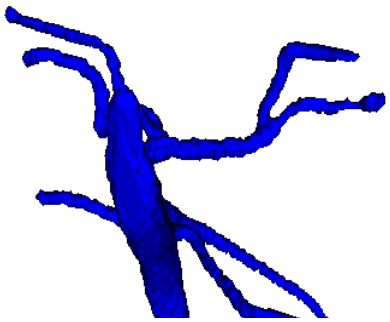


Figure 4.49: Original aorta mesh that needs to be smoothed.

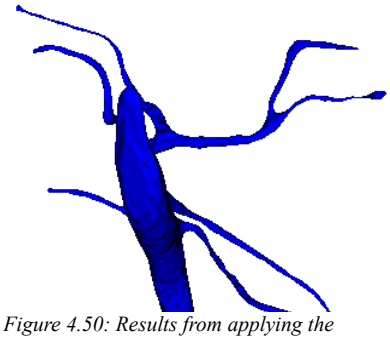


Figure 4.50: Results from applying the Laplacian Smooth Tool with 100 iterations and 0.1 relaxation factor. Notice the small vessels are much thinner than the original mesh.

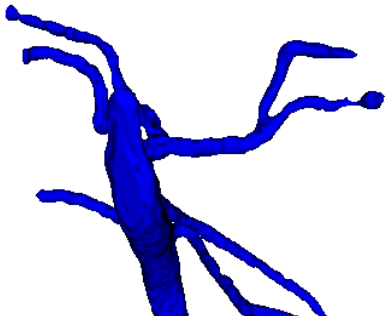


Figure 4.51: Results from applying the Windowed Sinc Tool with 100 iterations and 0.2 pass band. Notice the small vessels have not shrank as much as with Laplacian smoothing.

The Windowed Sinc Smooth Tool provides a smoothing alternative that better handles shrinkage of the mesh, as shown by Figure 4.51. This smoothing algorithm is provided by vtkWindowedSincPolyDataFilter [77] and is controlled similarly to the Laplacian smoothing: the number of iterations controls the amount of smoothing, and sharp feature edges can be preserved by specifying feature edge angles. A parameter unique to this algorithm is the pass band, which further controls the amount of smoothing. The filter is based on the frequency data of the mesh, so the pass band defines the maximum frequency that is allowed after smoothing. Since noise or bumps have high frequencies, they will be smoothed more by the filter, reducing the overall frequencies of the mesh to produce a better surface.

VTK's windowed sinc algorithm is based on Taubin's optimal surface smoothing method [72]. The windowed sinc filter is a standard signal processing low-pass filter and is used to smooth the surface similarly to smoothing and removing high frequency noise from an image. Chebyshev polynomials are used to approximate the filter's transfer functions, providing an iterative diffusion process to apply the filter. Each smoothing iteration applies the next higher term of the Chebyshev filter approximation, so more iterations will better approximate the filter transfer function. VTK's implementation avoids the severe scaling and translation caused by using too few iterations by modifying the pass band [77]. Numerical stability is also improved by translating and scaling the mesh to fit within the unit cube. After smoothing, the resulting mesh is rescaled and translated back to the original position.

The vertices are not the only part of the geometry that needs smoothing. The normals of the vertices should also be generated so that the surface appears smooth and continuous. Figures 4.52 and 4.53 illustrate how important the normals are to make a smooth surface. These normals are also used when calculating normal forces in simulations, so they are very important for calculating the correct force feedback for haptics devices. If the normals in a small region vary greatly, the surface will seem bumpy when it should actually be smooth. The Normals Tool uses the vtkPolyDataNormals [77] filter to generate the normals for a selected mesh. Several parameters allow the user to define how normals are created. An option allows the minimum angle of a sharp and feature edge to be defined so that these edges can be split. Splitting these edges allows different normals to be generated for each side of the edge so that it does not appear smooth. Consistent polygon order can be enforced while the normals are being generated so that the ordering of the vertices appears the same for each triangle from the outside of the mesh. The filter can be set to automatically orient all generated normals to point outward from the surface rather than inward, and if the normals are pointing in the wrong direction, they can be flipped. The filter generates normals either for the vertices or triangles, or both depending on the options. The algorithm works by first determining normals for each triangle using the cross-product of the edges. Then each vertex's normal is the average of the adjacent triangles' normals. If edge splitting is enabled, then sharp edges are split before averaging the normals at each point so that the correct normals are averaged together. The other options are applied by simply flipping normal directions when necessary.

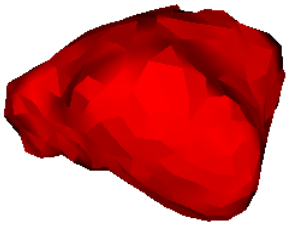


Figure 4.52: Liver mesh before generating normals. Each triangle strip is smooth but is easily distinguishable from other strips. This mesh contains 660 triangles.

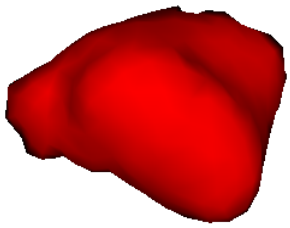


Figure 4.53: The liver mesh after generating normals with the Normals Tool. Now the surface appears continuous and smooth, but there are still 660 triangles.

4.5.3 Tetrahedralisation

The surface mesh alone is not sufficient for some simulations because the internal volume must be represented to perform mechanical calculations with mass-spring systems or the Finite Element Method. Therefore, a representation of the object's internal volume is necessary. In a majority of cases, a tetrahedral representation is chosen to fulfil this task. A tetrahedral mesh can be generated from a surface mesh with the Tetrahedralisation Tool. TetGen provides the functionality to create a tetrahedralisation suitable for simulations from a selected surface mesh [83]. The user can control the quality of the tetrahedra by setting the minimum radius-edge ratio, and a volume constraint can also be imposed on individual tetrahedra so that a desired resolution is generated. This level of control is necessary because the quality and size of tetrahedra greatly affect the accuracy and speed of simulation calculations. The splitting of the interior boundary (surface of the mesh) or exterior boundary (open side of the mesh) can be allowed or suppressed. If allowed, Tetgen can generate more points along the surface to create a better quality tetrahedralisation. However, exterior splitting must be disabled if the mesh will be later connected with another mesh portion of the same object. Like CGAL, the surface mesh must be saved as a temporary OFF file using the GeometryWriter and then loaded by TetGen into a tetgenio structure. After performing the algorithm, the resulting tetrahedra are accessed directly from the output tetgenio structure and stored in a TetMesh. A TetMesh SceneObject derives from the Mesh class to provide storage of 1D, 2D, or 3D cells in a vtkUnstructuredGrid [77], since these cells cannot be stored in vtkPolyData. The unstructured grid is converted to polygonal geometry by the vtkGeometryFilter for visualization and saving file types that do not support 3D cells.

Tetgen's algorithms build upon constrained Delaunay tetrahedralisation [73] to generate a tetrahedral mesh that fills the interior of the object and preserves the surface. Tetrahedralisation is similar to triangulation, except that each tetrahedron is made up of four points and has volume. The initial tetrahedra are formed by building a Delaunay tetrahedralisation from the surface points of the mesh. Because these tetrahedra will vary greatly in volume and radius-edge ratio, they will not meet the desired quality requirements. Therefore, a sequence of refinements are performed, flipping and splitting tetrahedra to add new vertices until the quality and size constraints are met. The mesh will be more refined with a smaller radius-edge ratio, providing better quality tetrahedra. Mesh refinement also causes the size of tetrahedra on the boundary of the object to be smaller than the interior. This range of sizes works well for simulations because interaction with the object will take place at the boundary where the smaller resolution improves the accuracy of the calculations. The main limitation of TetGen's algorithms is that the input boundary should not contain acute angles, or in practice, angles smaller than 60 degrees [83]. As long as the surface mesh meets this condition, a good quality tetrahedral mesh will be generated. Figure 4.54 shows the tetrahedral mesh generated from the liver surface mesh.

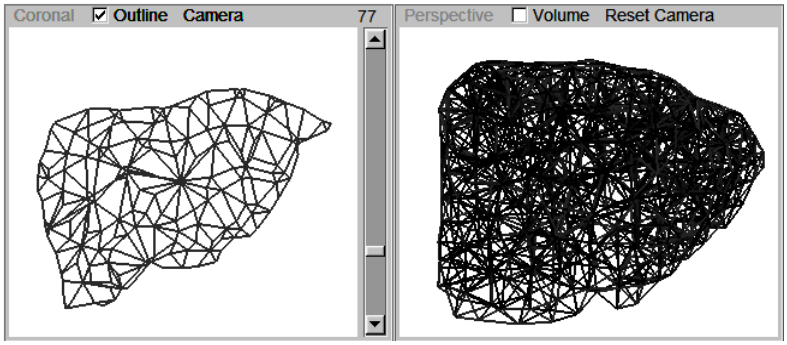


Figure 4.54: Tetrahedral mesh generated from the liver surface mesh with the Tetrahedralisation Tool.

5 - Experimentation

A series of experiments was conducted to test and evaluate the main algorithms used in the program. The overall goal was to provide a set of tools that can generate anatomical models from patient datasets. The ability to chain the tools together to accomplish this goal was analysed and demonstrated. The efficiency and accuracy of the segmentation methods was also tested. Results from the Hierarchical Segmentation were compared with other common segmentation methods. The abilities of the Hessian Vesselness filter were tested and extended by combining it with Fast Marching Segmentation to provide a fast method to segment vascular structures. Centrelines were generated from segmented vessel datasets, and their reconstructions were used to analyse their ability to represent the original structures.

5.1 - Mesh Generation from Patient Datasets

The program allows the user to easily perform a sequence of tools, each providing a step towards a final goal. One possible goal is to generate an anatomical model that is suitable for simulation. The following example sequence shows the chain of tools in order to generate a tetrahedral mesh of an aortic aneurysm from a small patient dataset.

The user must first load the patient dataset. Then preprocessing tools can be used to select a small region of interest or resample the image to reduce the amount of information. In this case, the dataset shown in Figure 5.1 is already small enough to proceed. This dataset is only 150 x 150 x 100 voxels, allowing segmentation to be performed very quickly.

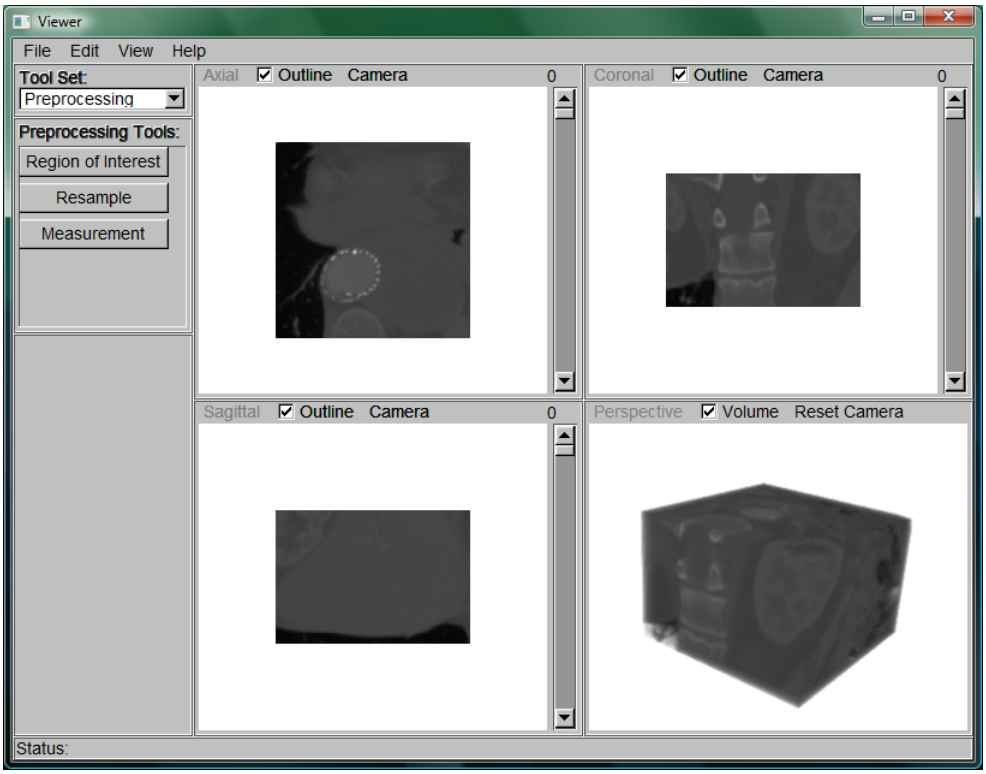


Figure 5.1: A small 150 x 150 x 100 voxel dataset has been loaded. The dataset contains an aortic aneurysm which will be segmented and meshed.

The user may wish to change the intensities of the dataset so that different structures are more visible. Figure 5.2 shows the Contrast Tool being used to make the structures surrounding the aorta invisible. By changing the contrast, the aorta is easily visible in the volumetric rendering. The mapped contrast values can also be used during segmentation so that unnecessary regions can be easily removed and the contrast at boundaries can be increased.

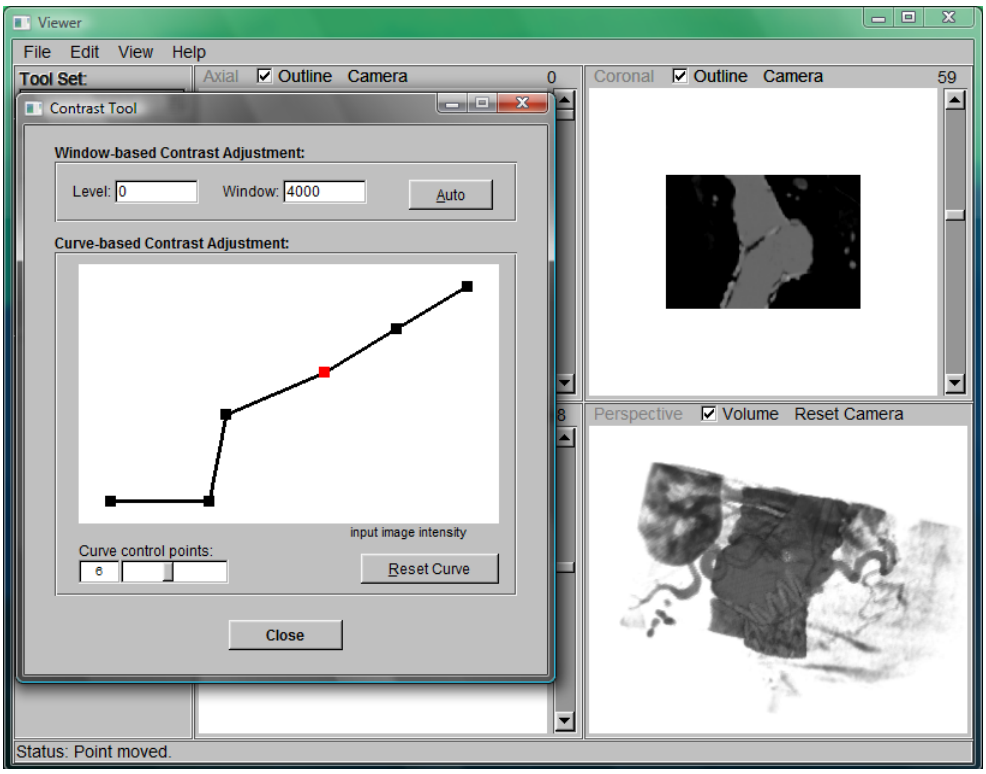


Figure 5.2: The Contrast Tool has been used to make the inner structure visible in the volumetric rendering.

The Hierarchical Segmentation Tool is then used to segment a series of slices, as shown in Figure 5.3. The segmentations can be interactively improved by marking more interior and exterior points. The interior points fall within the aorta, while the exterior points are placed in the surrounding region.

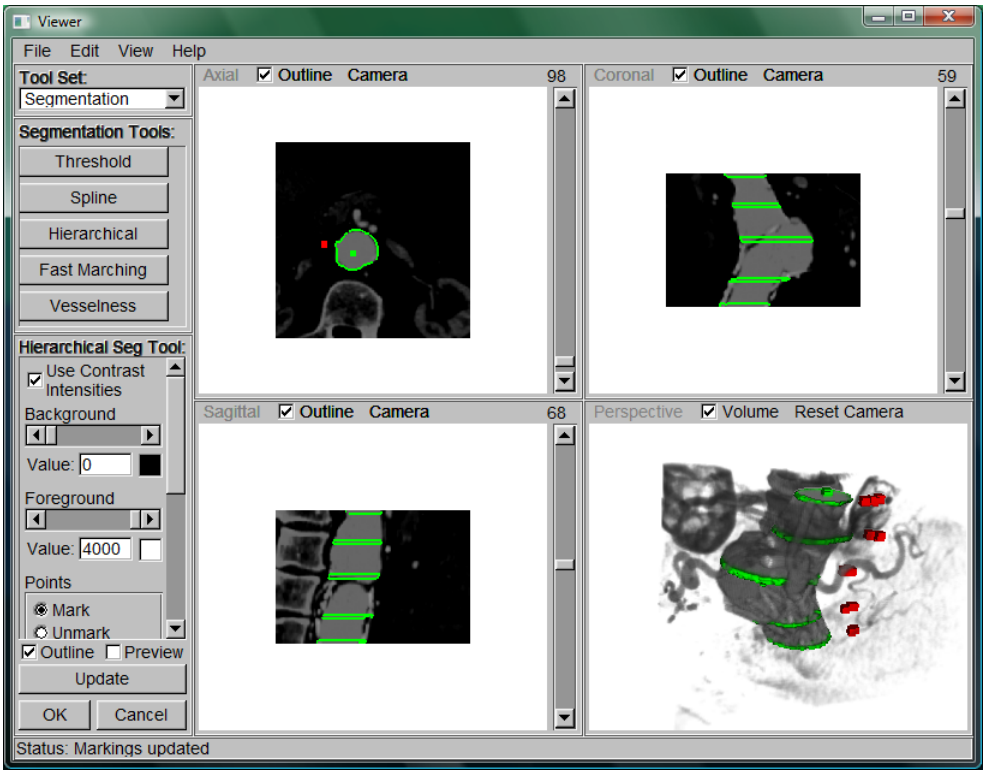


Figure 5.3: Several slices of the dataset have been segmented (in green) with the 2D mode of the Hierarchical Segmentation Tool. The red points indicate exterior markings, while the green points are interior markings. The mapped values from the Contrast Tool are used to remove unnecessary structures, making segmentation quicker.

The individual 2D Hierarchical segmentations can then provide the markings for extension into a complete 3D segmentation. Figure 5.4 shows the resulting 3D segmentations after automatically

extending the segmentations in each sagittal slice.

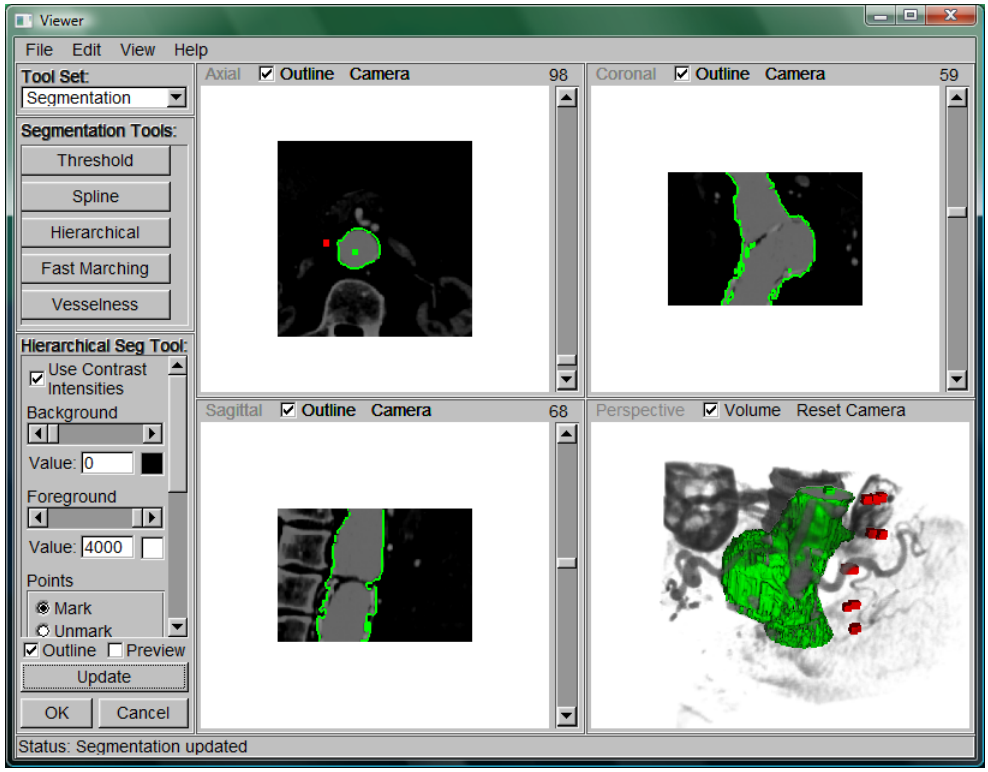


Figure 5.4: The individual 2D Hierarchical Segmentations have been extended to generate a complete 3D segmentation, as shown by the green outline.

Now that the dataset has been segmented, a triangular surface mesh can be generated via CGAL's algorithms with the Surface Mesher Tool. Figure 5.5 shows the parameters being set, with the sphere surrounding the region to be meshed. The resulting mesh is visible in Figure 5.6.

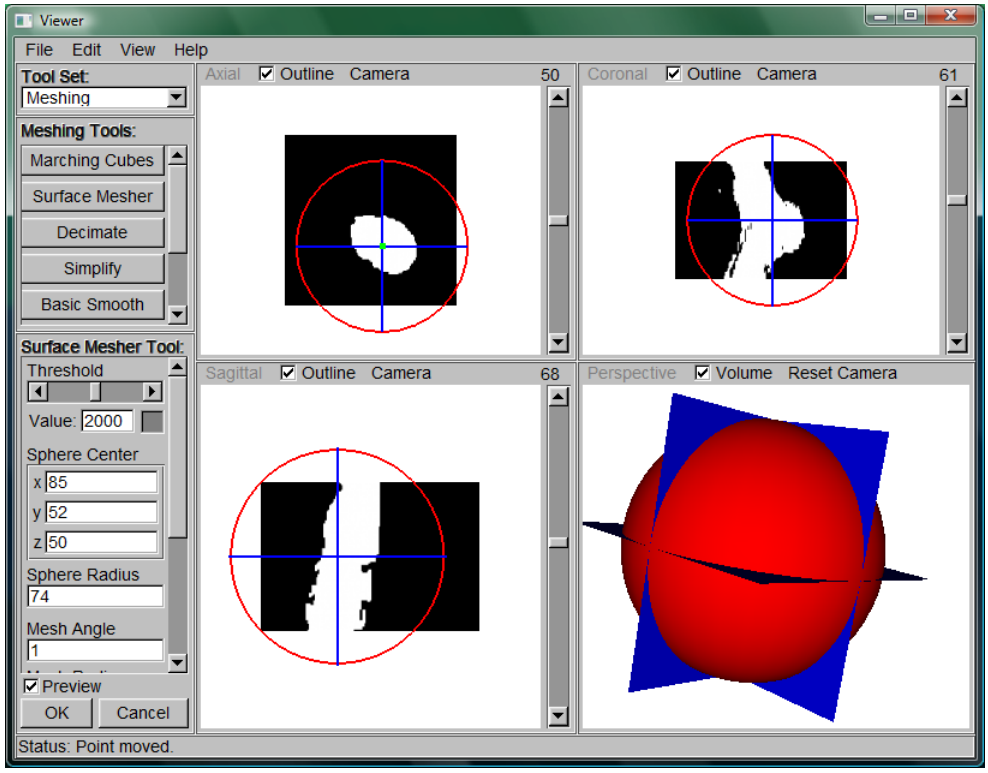


Figure 5.5: The Surface Mesher Tool is being used to generate a mesh of the aorta's surface. The centre of the sphere is placed within the segmented structure, and the radius is set so that the sphere surrounds the entire segmented region.

The resulting surface mesh has some unwanted sharp edges, which need to be smoothed. Applying the Laplacian smooth will cause shrinkage, so the Windowed Sinc smoothing is used. As seen in Figure 5.6, The pass band is set high (0.5) so that most of the major features are not lost. The resulting mesh can be seen in Figure 5.7.

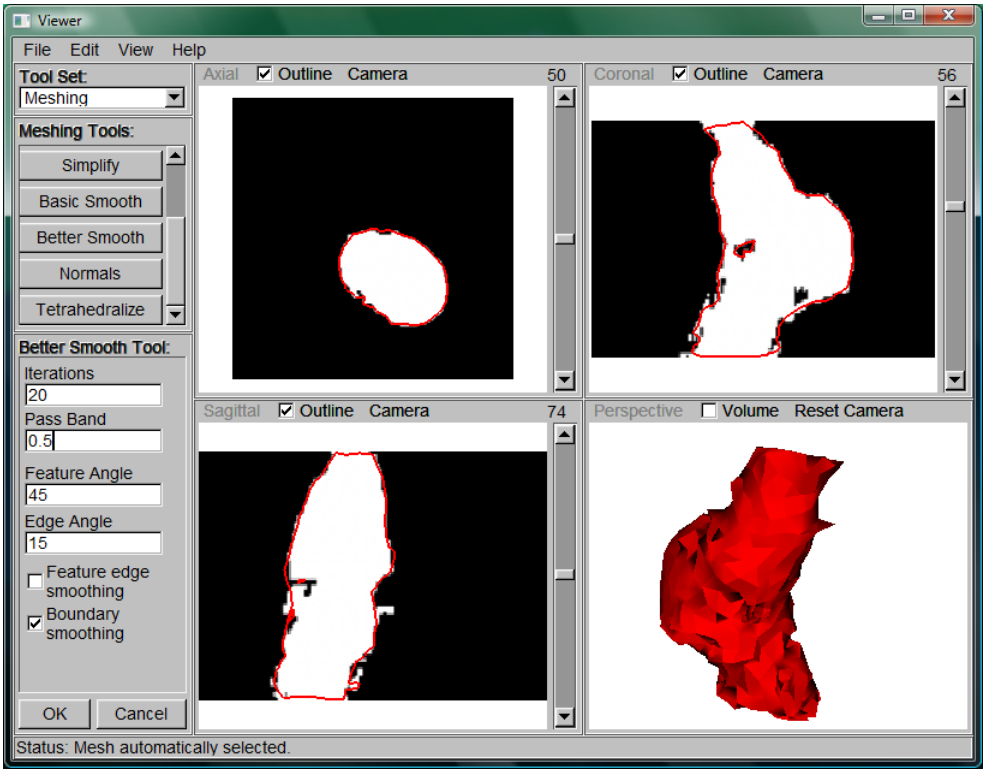


Figure 5.6: The Windowed Sinc Smooth Tool is being used to perform smoothing without shrinking the mesh. The pass band is chosen so that important features are not lost.

The individual triangles of this surface mesh are easily visible because the normals are simply perpendicular to the triangle faces. The normals can be regenerated with the Normals Tool to provide a smooth appearance, as shown in Figure 5.7. These normals are not only important for visualisation, but they also are needed to generate the correct force feedback for haptics devices during simulations. Figure 5.8 shows the surface with smooth normals. Note that this does not affect the actual geometry: the number of triangles is still the same.

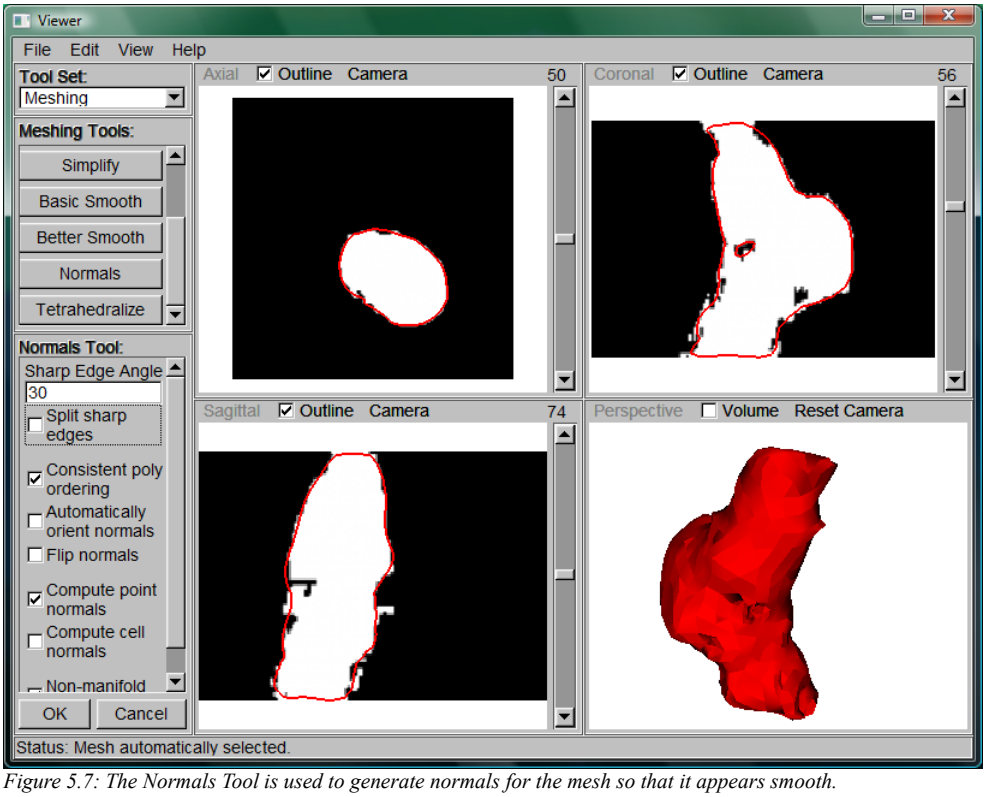


Figure 5.7: The Normals Tool is used to generate normals for the mesh so that it appears smooth.

The mesh so far only represents the surface of the aorta. For the finite element method or mass-spring systems used in simulations, a representation of the inner volume is necessary. A tetrahedral mesh, generated by the Tetrahedraliser Tool using the TetGen library will generate such a representation. Figure 5.8 shows the default options for tetrahedralisation that were used in this case to generate the quality tetrahedral mesh in Figure 5.9.

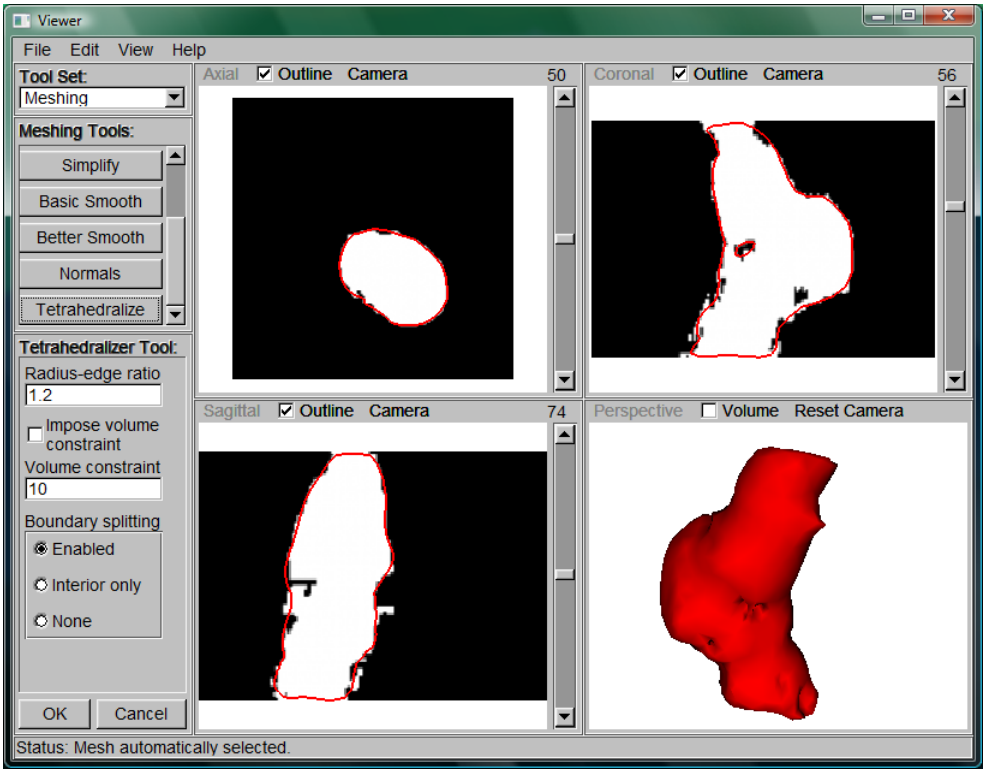


Figure 5.8: A tetrahedral representation of the object is created from the surface mesh with the Tetrahedraliser Tool.

This final tetrahedral mesh in Figure 5.9 can now be saved to a geometry file and used in

simulations. These tetrahedra discretise the volume of the object for the finite element method. Alternatively, the vertices can act as masses and the edges as springs in a mass-spring system. These simulation methods allow the user to interact with deformable objects in real-time.

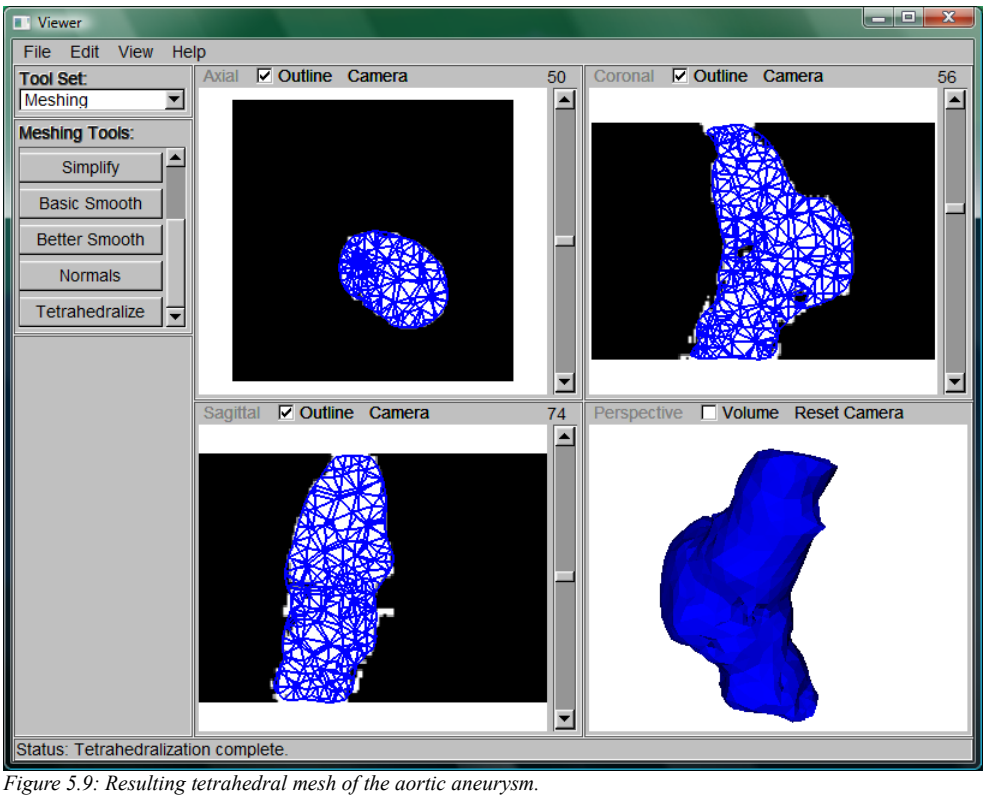


Figure 5.9: Resulting tetrahedral mesh of the aortic aneurysm.

Different mesh generation algorithms will produce slightly different results. Additionally, decimation and smoothing will cause these results to deviate further from the original segmentation. The minimum, maximum, mean, and root mean square (RMS) Hausdorff distances provide a comparison of how similar two surface meshes are to each other. These distances were measured using MESH [84], a free comparison software, to determine the effects of algorithms on the accuracy of the mesh.

Figure 5.10: Minimum, maximum, mean, and root mean square Hausdorff distances between the mesh of the liver generated by CGAL to the result from the Marching Cubes Algorithm.

The resulting mesh from the Marching Cubes algorithm and CGAL's surface mesher were compared to determine how close CGAL's generated meshes are to the segmented dataset. The Marching Cubes algorithm will produce a mesh which very closely follows the segmentation, providing a reference mesh. CGAL's surface mesher allows control over the resolution of triangles, and large triangles will not match the original segmentation as closely as the small triangles of a Marching Cubes mesh. The CGAL surface seen in Figure 5.11 was generated with a mesh angle of 1, mesh distance of 10, and mesh radius of 10. The distances are greater around sharp edges, rising to around 2 mm. The samples chosen by

CGAL usually miss these thin areas when the mesh radius is set larger than their thickness, as seen here. As shown in Figure 5.10, the minimum Hausdorff distance was 4.19 x 10^{-5} mm, the maximum was 42.16 mm, the mean was 4.34 mm, and the RMS distance was 8.3 mm. Although CGAL's mesh is better quality, with more regular triangles, the difference can be large in some areas.

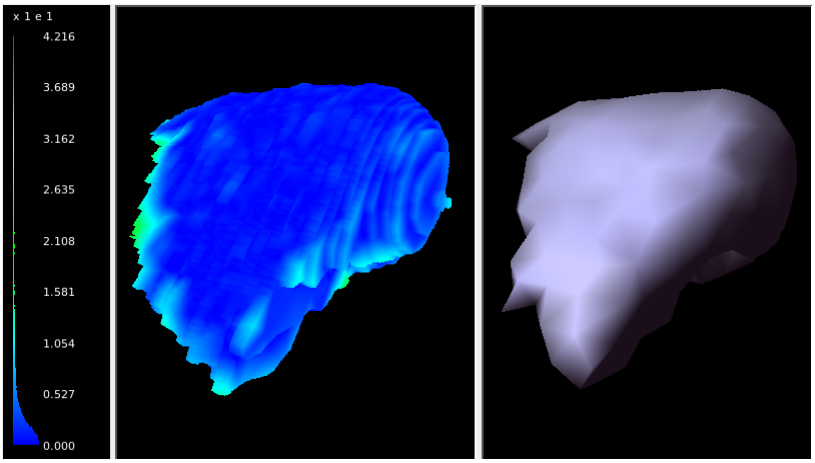


Figure 5.11: Comparison of the CGAL-generated mesh and the result from the Marching Cubes Algorithm. Lower Hausdorff distances are indicated by darker colours (blue), while brighter colours (green, yellow, and red) are higher Hausdorff distances.

To determine the effect of decimation on the accuracy of a mesh, the Hausdorff distances were measured between the original Marching Cubes mesh and decimated mesh. A target reduction of 0.9 was used to reduce the number of triangles by 90%. Topology was preserved to prevent holes, but splitting was enabled so that the desired reduction could be accomplished. The original liver mesh has 263,660 triangles, while the decimated mesh has 26,366 triangles. Figure 5.12 shows the minimum, maximum, mean, and RMS Hausdorff distances. The minimum is 0 mm, the maximum is 3.45 mm, the mean is 0.23, and the RMS is 0.36. These distances are small, showing that even with such a large reduction, a decimated mesh closely matches the original mesh.

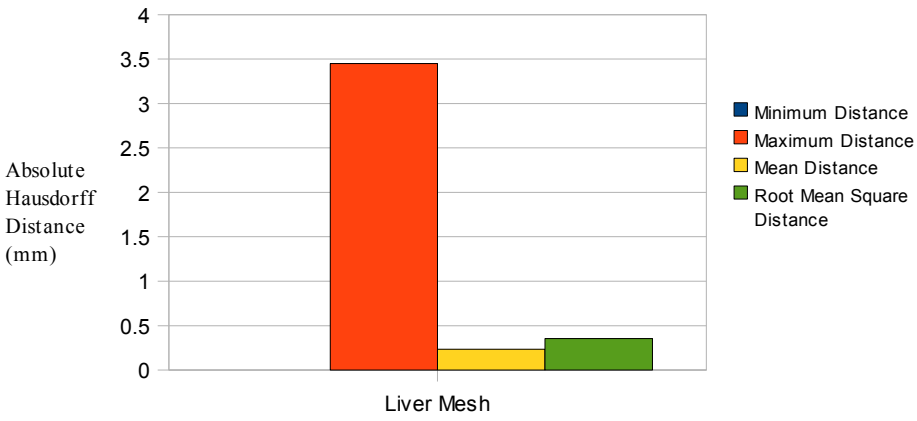


Figure 5.12: Minimum, maximum, mean, and root mean square Hausdorff distances between the Marching Cubes liver mesh to the decimated mesh. The blue bar is not visible because the minimum distance is 0 mm.

As seen in Figure 5.13, the Hausdorff distances tend to be larger in the flat areas of the original mesh. The number of triangles is reduced the most in these areas because their contribution to the overall shape is less than sharper or bumpier areas. As triangles are removed, some local volume is inevitably lost, leading also to more variation from the original mesh.

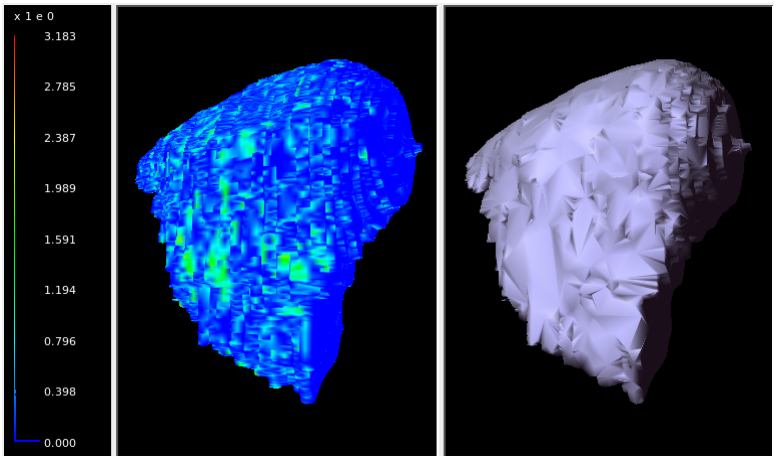


Figure 5.13: Hausdorff distances comparing the Marching Cubes mesh to the decimated mesh.

Another experiment determined the difference in the amount of shrinkage between Laplacian smoothing and Windowed Sinc smoothing. A mesh of the aorta and smaller connected vessels was smoothed with both algorithms. The Laplacian smooth was performed with 200 iterations and a relaxation factor of 0.1. The Windowed Sinc smooth was performed with 200 iterations and a pass band of 0.05. A low pass band was chosen to provide a similar amount of smoothing compared to the Laplacian results. Figure 5.14 compares the minimum, maximum, mean, and RMS Hausdorff distances for the two smoothing methods. Windowed Sinc's maximum Hausdorff distance of 2.76 mm is smaller than the Laplacian's 3.47 mm. The mean of the Windowed Sinc's results is 0.45, lower than Laplacian's 0.77. Windowed Sinc's RMS Hausdorff distance is also smaller than the Laplacian's, 0.53 compared to 0.9. These distances show the results from Windowed Sinc smoothing to be closer to the original mesh than the Laplacian smoothed mesh.

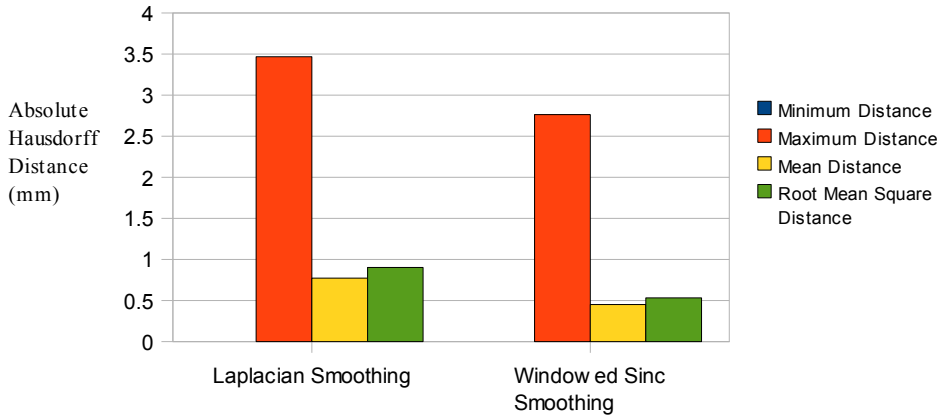


Figure 5.14: Comparison of the minimum, maximum, mean, and root mean square Hausdorff distances for Laplacian smoothing and Windowed Sinc smoothing. The blue bars are not visible because the minimum distances are 0 mm.

Figures 5.15 and 5.16 compare the Hausdorff distances along the surface of the vessels. As expected, the Laplacian smoothing causes the smaller vessels to have greater distances than Windowed Sinc smoothing. However, Windowed Sinc smoothing still causes shrinkage at the very thin tips of some vessels. These tips have such small radii that they fall above the pass band. The pass band can be raised to reduce the amount of shrinkage, but this will also reduce the amount of smoothing.

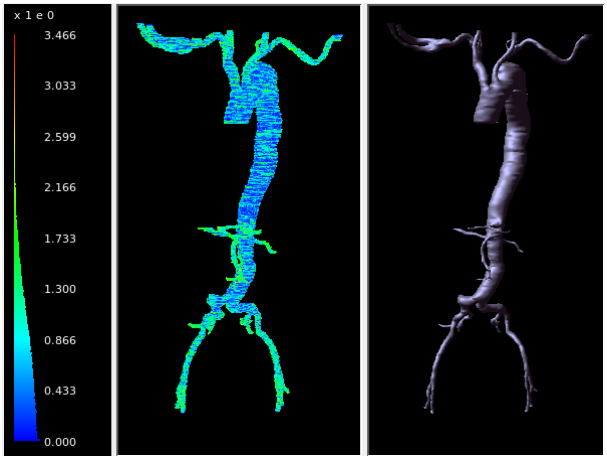


Figure 5.15: Hausdorff distances comparing the Marching Cubes vessel mesh to the Laplacian smoothed mesh.

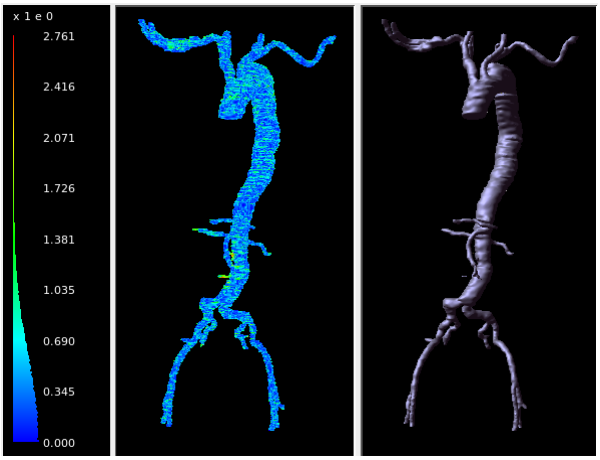


Figure 5.16: Hausdorff distances comparing the Marching Cubes vessel mesh to the Windowed Sinc smoothed mesh.

5.2 - Hierarchical Segmentation

The Hierarchical Segmentation provides a fast, interactive segmentation algorithm. Below is a simple example of its use for both 2D and 3D segmentations of datasets. Figure 5.17 shows the small dataset that was used: its dimensions are 150 x 150 x 100 voxels. The object of interest within the dataset is an aortic aneurysm, highlighted in red.

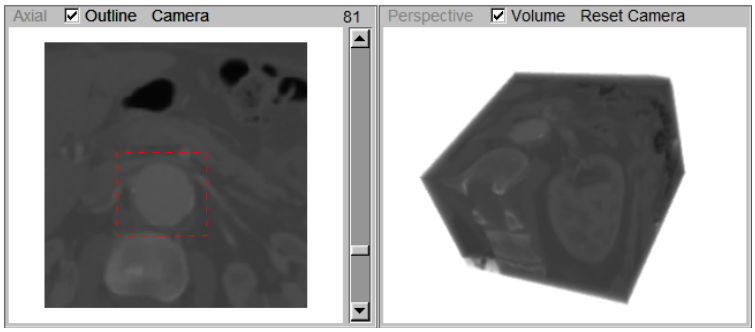


Figure 5.17: Original dataset. In this example, the aortic aneurysm (red) will be segmented.

A single 2D slice can be segmented very quickly by marking interior and exterior points, as shown in Figure 5.18. The segmentation for this slice was generated in about 1 second, providing the user with the ability to interactively segment the structure. In this case, only two marking points were required - one interior and one exterior - to segment the slice. In cases with less contrast between regions, more points can be placed to improve the segmentation.

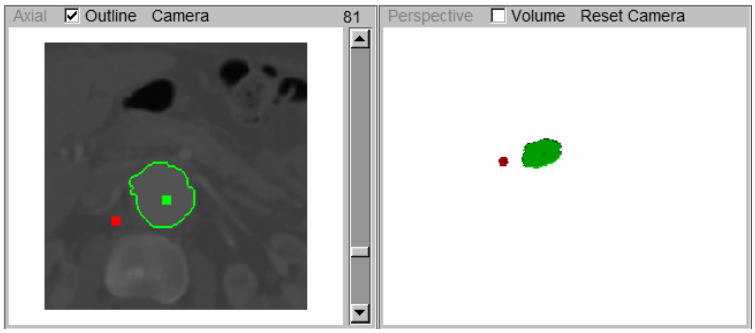


Figure 5.18: A single 2D hierarchical segmentation in an axial slice. The red point marks the exterior, while the green point marks the interior of the segmented structure. The green outline shows the boundary of the resulting segmentation.

Many 2D slices may be segmented individually, as shown in Figure 5.19. The user could fully segment the dataset one slice at a time in this manner. Alternatively, a 3D segmentation can be performed. The Hierarchical Segmentation Tool provides two methods for generating a 3D segmentation. The first is

a true 3D segmentation, using the Hierarchical Segmentation algorithm on the entire dataset. The second is a faster algorithm which extends the user's 2D segmentations into 3D.

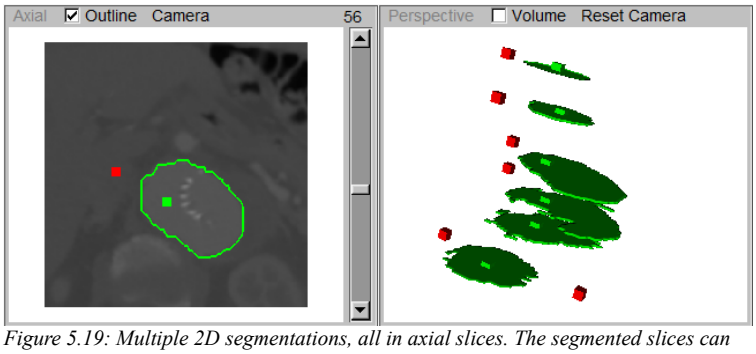


Figure 5.19: Multiple 2D segmentations, all in axial slices. The segmented slices can be easily seen in the perspective view, along with the interior and exterior markings.

The interior and exterior markings can be used to perform the full 3D segmentation on the entire dataset. However, this process is very slow and gives poor results as seen in Figure 5.20. A 3D segmentation was attempted on this small dataset using 30 markings; the resulting segmentation required 10.5 minutes to generate. Many more points and much more time will be required to generate a good quality segmentation. In a 3D segmentation, the amount of information is much greater, with each region having many neighbours. This complexity not only requires more memory and more time, but it also leads to sub-optimal region merging.

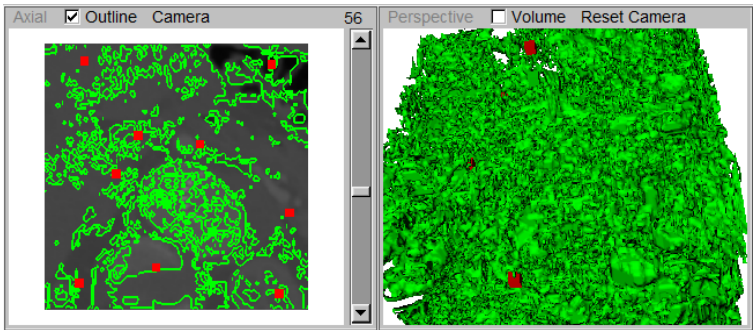


Figure 5.20: Attempted 3D segmentation using 30 markings. The extra dimension over 2D causes each region to have many more neighbours, often causing sub-optimal merge operations. This segmentation required 10.5 minutes to generate.

The alternative extension of the 2D segmentations to a 3D segmentation reduces the complexity by working in 2D. The perpendicular extending slices gain many markings from intersections with the user's 2D segmentations, leading to a good 3D segmentation. This algorithm is also faster, taking only 4.5 minutes and 12 markings to generate the segmentation in Figure 5.21. The user can also easily improve this 2.5D segmentation by improving and adding more 2D segmentations.

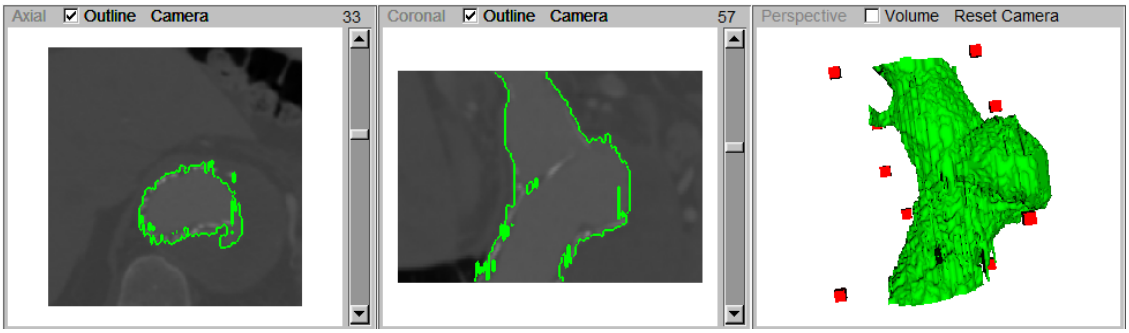


Figure 5.21: Results from extending the 2D hierarchical segmentations shown in Figure 5.3 into 3D. This segmentation required only 12 marked points and only 4.5 minutes to generate.

To test the effectiveness of the Hierarchical Segmentation application in extracting vascular structures, Nizar Din segmented 20 patient datasets, including 19 CTA and one MRA scans [85]. Manual segmentation and ITK-Snap level set segmentations were also performed on each dataset for comparison. Three measures were analysed: segmentation time, symmetrical mean Hausdorff distance between each

of the vasculatures and the corresponding manual segmentation, and minimum Hausdorff distance under which most of the surface nodes reside. MESH [84] was used to compute the Hausdorff distances.

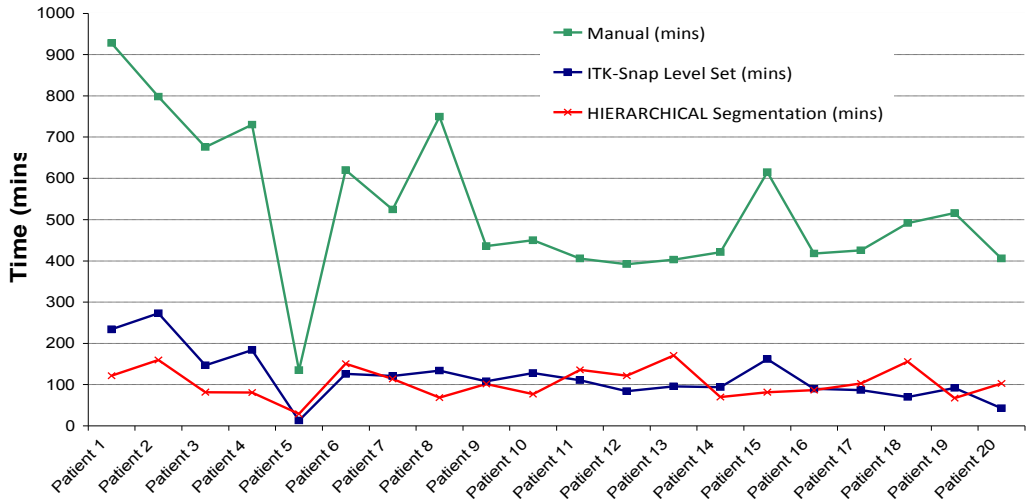


Figure 5.22: Time required to segment each patient dataset. The green line shows manual segmentation, the blue line shows ITK-Snap's level set segmentation, and the red line shows Hierarchical Segmentation.

The manual segmentation was most time consuming, taking 527.5 minutes on average. Figure 5.22 shows the manual segmentation time gradually decreases as the operator becomes accustomed to the software. Patients 8 and 15 required more time because of technical problems (ITK-Snap crashing) due to the size of the dataset. Patient 5 required much less time for all methods because it was the MRA scan with only 385 slices compared to about 600 for the CTA scans. ITK-Snap's level set segmentation had a mean time of 119.5 minutes, about 77% less time than manual segmentation. The Hierarchical Segmentation was slightly faster than the level set method, with an average time of 104.2 minutes, which is about 80% less than manual segmentation.

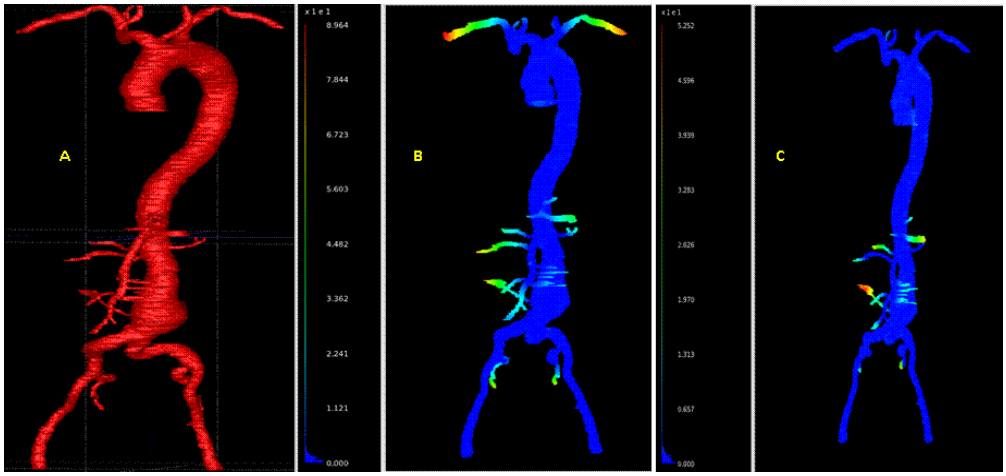


Figure 5.23: Comparison, using MESH, of the segmentations for one dataset. Darker colours (blue) represent smaller Hausdorff distances, while lighter colours (green, yellow, and red) indicate larger Hausdorff distances. A shows the manual segmentation, B shows the ITK-Snap level set segmentation, and C shows the Hierarchical Segmentation.

The symmetrical mean Hausdorff distance and the minimum Hausdorff distance were used to verify the Hierarchical Segmentations by using the manual segmentation as a reference. Figure 5.23 compares the Hausdorff distances for both the level set and Hierarchical segmentations of a single dataset. The main difference between the segmentations is the small vessels connected to the aorta. These small structures are not easily segmented by either level sets or Hierarchical segmentations and are therefore leading to larger Hausdorff distances at their locations. The user must usually place more seeds or marked points in these regions to generate a complete segmentation. Manual segmentation of small vessels is also difficult, causing variation.

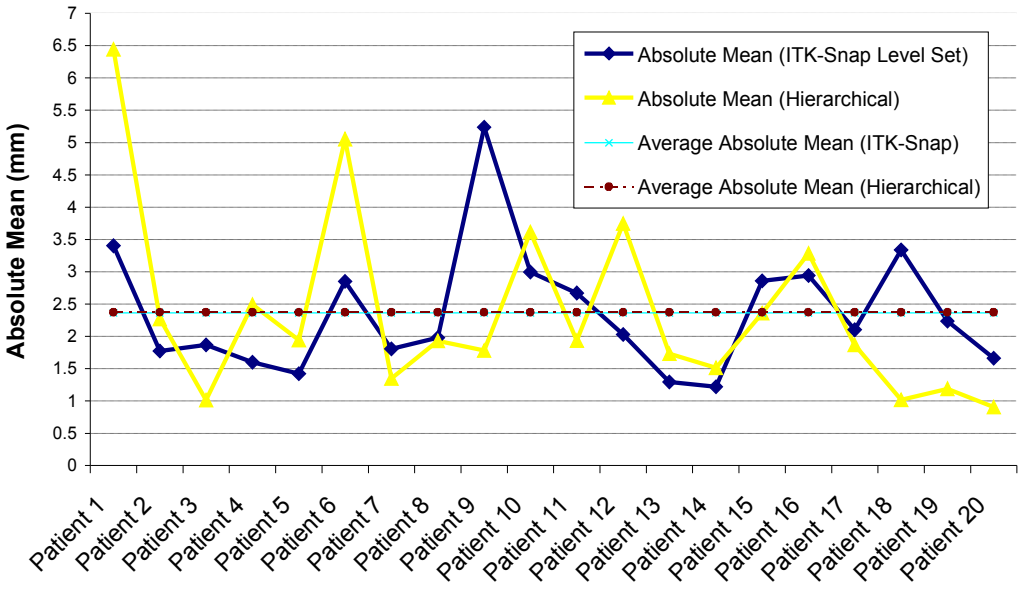


Figure 5.24: Comparison of the absolute mean Hausdorff distances for each dataset. The dark blue line shows the mean distances for the ITK-Snap level set segmentations, while the yellow line shows the mean distance for the Hierarchical Segmentations. Both averages were slightly less than 2.5 mm.

The symmetrical mean Hausdorff distances for the segmentation are displayed in Figure 5.24. The distance for the ITK-Snap level set segmentation varies from 1.22 mm to 5.24 mm. The average symmetrical mean Hausdorff distance from the manual segmentation was 2.36 mm, while the root mean square (RMS) error was 7.45 mm. For the Hierarchical Segmentation, the symmetrical mean Hausdorff distance varies from 0.904 mm to 5.24 mm. The average distance of 2.37 mm compares well with that of the level set segmentation, with an RMS error of 5.99 mm.

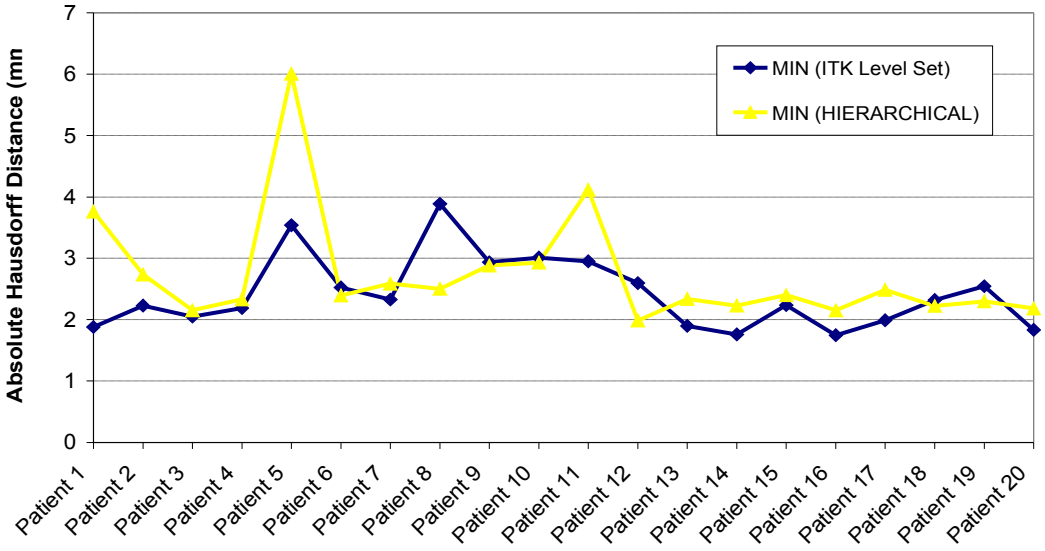


Figure 5.25: Comparison of the minimum Hausdorff distance for each segmentation. The blue line corresponds to the ITK-Snap level set method, while the yellow line is the Hierarchical Segmentation.

Figure 5.25 shows the minimum Hausdorff distances for each segmentation. This value is the minimum distance containing at least 90% of the surface nodes. ITK-Snaps level set method produced an average minimum Hausdorff distance of 2.42 mm, while it was 2.74 mm for the Hierarchical Segmentation method.

Overall, the minimum for Hierarchical Segmentation is only 0.32 mm larger, but Figure 5.6

showed that it takes on average 15 minutes less than ITK-Snap's level set method and around 7 hours less than manual segmentation. Therefore, Hierarchical Segmentation provides a quick method to segment detailed vasculatures with comparable accuracy to level set methods.

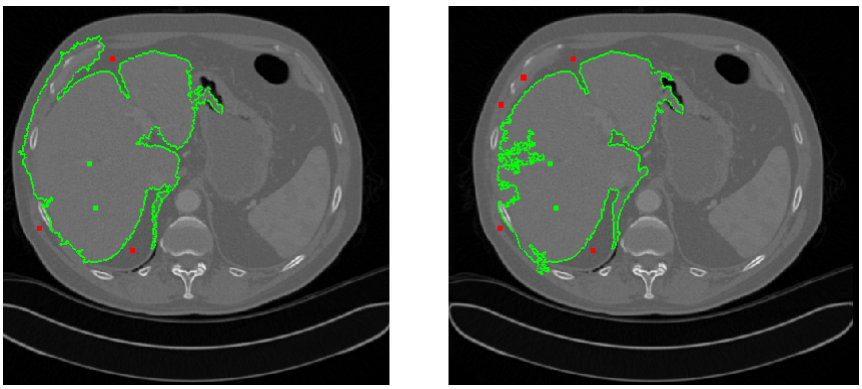


Figure 5.26: As more interior and exterior points are marked, the boundary of the segmented liver fluctuates. This requires a large number of marked points to generate an accurate segmentation.

Because Hierarchical Segmentation merges regions based on the contrast between their mean values, datasets with little or no contrast between the desired object and surrounding regions will pose difficulties. The user will be required to place a large number of points to clearly define the boundary of the structure; however, this process may be longer than manually delineating the boundaries of the object. An example of an organ which will often not have enough contrast with the surrounding regions is the liver.

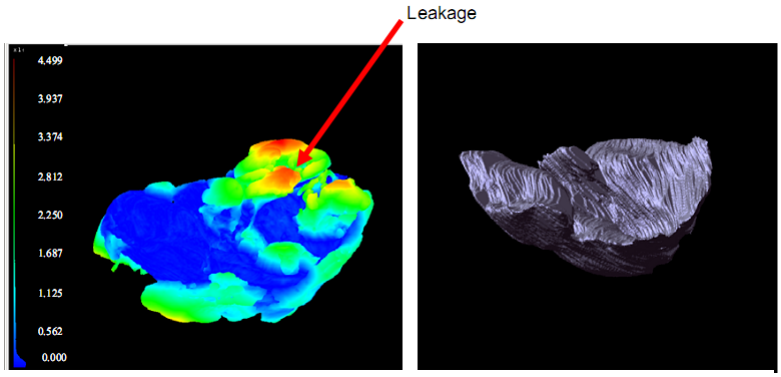


Figure 5.27: The Hausdorff distances of ITK's level set method show leakage in the segmentation in areas of low contrast.

To test the effectiveness of the Hierarchical Segmentation with liver datasets, Laura Tucker segmented one CT scan and one MRI [86]. The times and results were compared with those of manual segmentation an ITK-Snap's level set segmentation. Figure 5.28 shows the times to segment each dataset with both manual and Hierarchical segmentations. The low contrast between the liver and surrounding regions contributed to the Hierarchical Segmentation took up to 6 times longer than manual segmentation. The boundary of the segmentation was unstable as more points were marked, shifting back and forth to include or exclude more regions. Thus, more points had to be placed to ensure the boundary remains accurate, as shown in Figure 5.26. Level set segmentation was the fastest, taking only 12 minutes for the entire liver in dataset RLH004; however, the low contrast also affects it by causing the segmented region to leak out of the liver and into the surrounding regions, as shown in Figure 5.27.

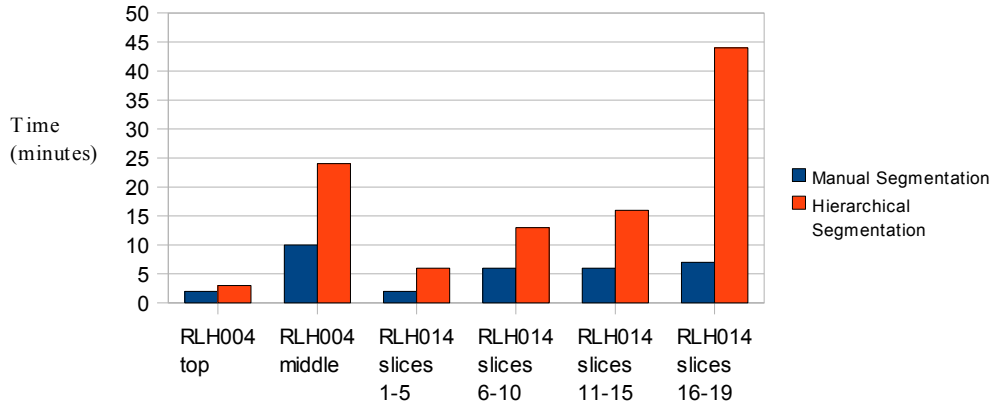


Figure 5.28: Comparison of segmentation times for each dataset. Because of low contrast and ambiguous boundaries, the Hierarchical Segmentation took much longer than manual segmentation.

The minimum, maximum, mean, and root mean square Hausdorff distances are shown in Figure 5.29. Although the level set method was very fast, it sacrifices accuracy with its maximum distance more than twice the maximum distance for the Hierarchical Segmentation of the same dataset. The Hierarchical Segmentation provides more accurate results, but at a high cost of time. A manual segmentation will give better results in less time; therefore, Hierarchical Segmentation is not useful for low contrast livers.

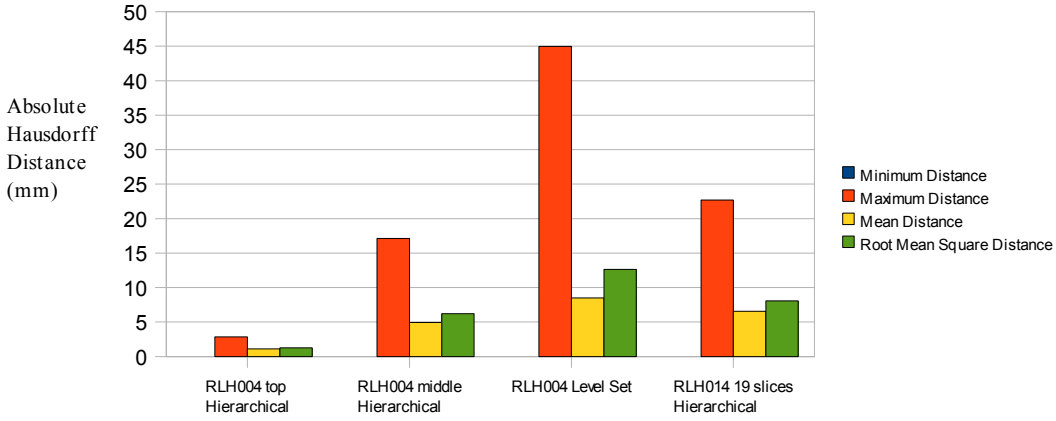


Figure 5.29: Minimum, maximum, mean, and root mean square Hausdorff distances for each segmentation. The level sets distances are significantly higher than those from Hierarchical Segmentation. The blue bars are not visible because the minimum distances are 0 mm.

5.3 - Hessian Vesselness Segmentation

To determine the effectiveness of the Hessian Vesselness tool to segment vascular structures, two patient datasets were selected. These datasets contain the aorta and other connected blood vessels (such as the renal and iliac arteries). An entire dataset could not be segmented at one time because of the memory requirements to store the Hessian matrix. Using the Region of Interest Tool, each dataset was first divided into three equal parts (Figure 5.30). The Hessian Vesselness Tool requires the minimum and maximum radius of the blood vessels in the image, so the Measurement Tool was used to measure these values. The Vesselness was then calculated for each part of the dataset, as shown in Figure 5.31.

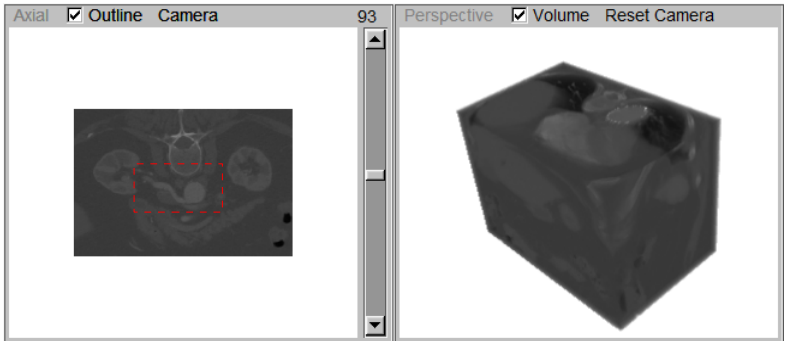


Figure 5.30: One part of a dataset to be segmented. The aorta and connecting blood vessels (red) will be segmented.

The Hessian Vesselness filter was applied only once to each part, using the measured minimum and maximum radii. The amount that the radius was increased each iteration was chosen to keep the number of iterations small, yet to still capture most of the structure. All regions with vesselness less than 0.9 were removed in each iteration. This high threshold ensures only the most vessel-like structures remain. Table 5.1 shows the radius and threshold values for each part of the two datasets: STM001 and STM007.

	Minimum Radius	Maximum Radius	Step	Threshold
STM001 slices 0-196	3	13	5	0.9
STM001 slices 197-393	3	18	5	0.9
STM001 slices 394-589	3	18	5	0.9
STM007 slices 0-197	3	13	5	0.9
STM007 slices 198-395	2	18	4	0.9
STM007 slices 396-593	2	20	6	0.9

Table 5.1: Minimum and maximum radii, steps, and threshold are the four parameters of the vesselness filter applied to the two dataset of this study: STM001 and STM007.

Datasets often contain additional tubular structures, all of which will be found by the Hessian Vesselness filter. Many of these structures will not be desired, such as parts of the vertebrae, parts of organs such as kidneys, and blood vessels which are not connected directly to the aorta. Figure 5.31 shows how many extra structures are segmented and must be removed. Because all of the desired structures are connected to the aorta, the Fast Marching Segmentation Tool can be used to grow seed points to fill these structures and avoid the other structures.

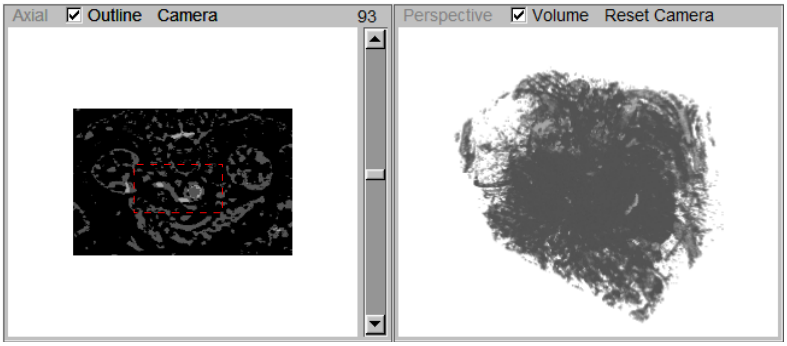


Figure 5.31: Results after applying Hessian Vesselness filter. All tubular structures have been segmented, many of which are not desired.

Before applying Fast Marching segmentation, the Contrast Tool is used to ensure all vessel-like structures have the same intensity. If this step is not done, then the varying intensity values of the vesselness measures will prevent the segmentation from filling the entire aorta. The vesselness measures can vary from 0 to a number higher than 1 during each iteration, and the final combined segmentation is scaled to yield values between 0 and the maximum unsigned short value (65535). This contrast also ensures high gradients between the vessels and surrounding regions, providing an excellent speed image for the Fast Marching filter. The speed image will allow the region to grow within the structure but be contained by boundaries. Now, the aorta and connecting vessels can be quickly extracted with a few seed points. By applying Fast Marching segmentation on the vesselness result, the user can choose which structures are desired and discard the others by placing seeds only within desired regions. Figure 5.32

shows the result of applying Fast Marching segmentation to the vesselness shown in Figure 5.31. The aorta and connected vessels are extracted and all other tubular structures discarded.

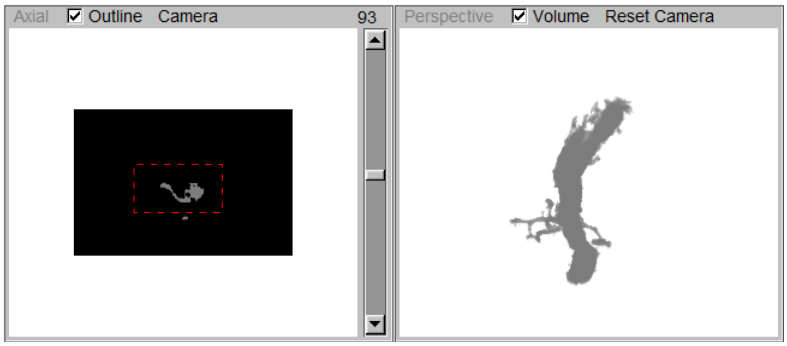


Figure 5.32: Results of Fast Marching Segmentation. In this case, 5 seeds were placed within the aorta and grown to fill the entire aorta and other connected blood vessels. Other unwanted structures are no longer present.

Because the gradient of the vessels boundary was high, the minimum contour gradient for the Fast Marching segmentation was set to 5000 (for example, the minimum gradient on the aorta boundary in the original dataset is around 400). The region growing was set to run 100 iterations for each dataset. The number of seeds required to fill the structures was small, as shown by Figure 5.33. The average number of seeds was 7.33. Slices 197-393 of STM001 required more seeds (21) because the output from the Hessian Vesselness filter was fragmented. Seeds had to be placed in each fragment to extract the entire structure.

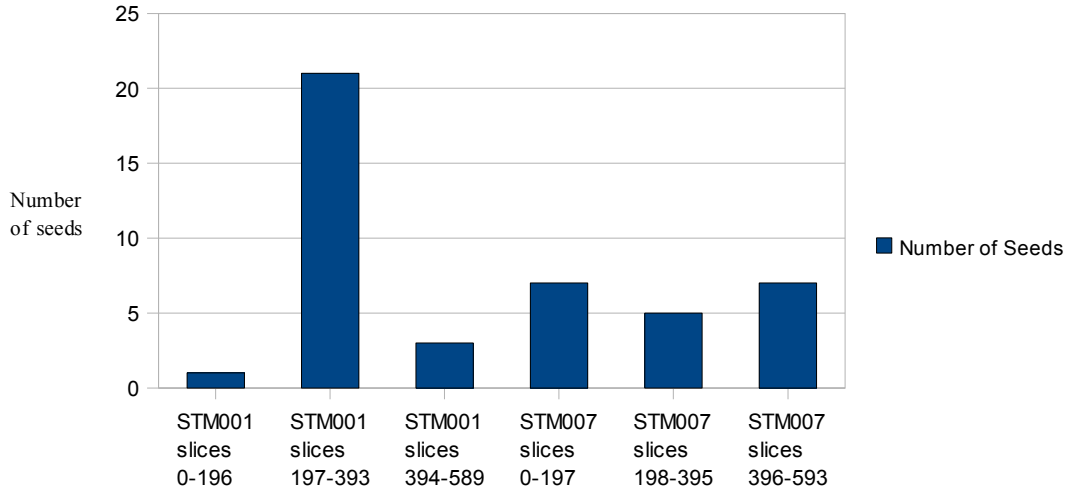


Figure 5.33: Number of seeds used for Fast Marching Segmentation of the Vesselness output for each dataset.

Bright structures, such as stents or calcification, cause problems in the vesselness measures because only tubular structures that are brighter than their surroundings are considered vessel-like. These bright structures cause the actual vessels to be darker than part of their surroundings, and so they are discarded. Figure 5.38 shows the resulting mesh of the fragmented vessels. To solve this problem, a threshold can be applied to remove the brighter areas before applying the Hessian Vesselness filter. Figure 5.39 shows how a threshold can reduce fragmentation and lead to better connected vessels. Figure 5.34 shows the original dataset with bright areas surrounding the aorta, while Figure 5.35 shows the result of applying a threshold to remove these unwanted structures. Figure 5.36 and 5.37 compare the effect on the vesselness of the aorta.

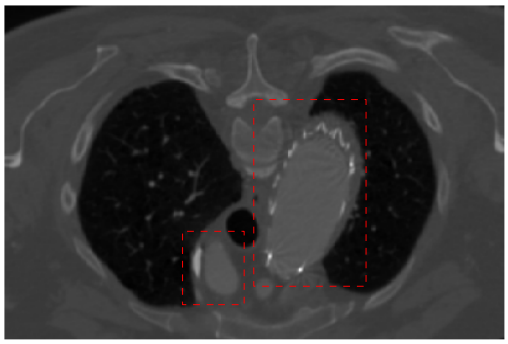


Figure 5.34: Original dataset with bright structures surrounding the aorta (red).

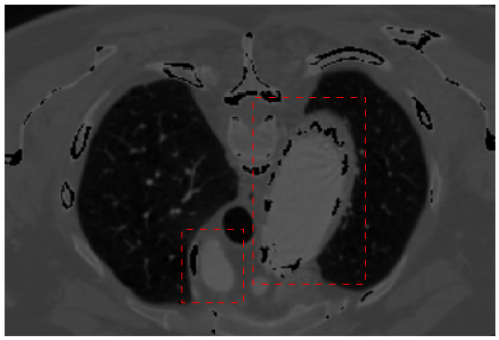


Figure 5.35: Dataset after removing bright structures with a threshold.

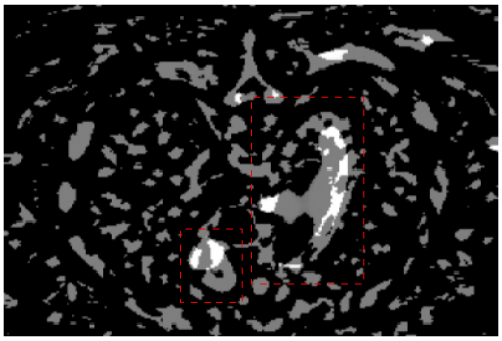


Figure 5.36: Vesselness of the original dataset. Notice the fragmentation of the aorta (red).

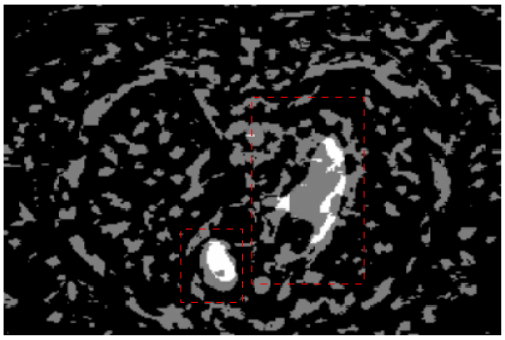


Figure 5.37: Vesselness of the dataset after thresholding. The aorta (red) is still fragmented, but not as much as before.

After extracting the vessels from each part of the dataset, the Marching Cubes algorithm was used to generate a triangular mesh of the structures. These meshes were combined into a single mesh of the entire aorta and many connected vessels, as shown in Figure 5.38 and 5.39. The segmentations from this experiment contain many more small vessels (especially around the aorta arch) than may be desired, but the Hessian Vesselness filter allows the user to ignore small vessels by setting a larger minimum radius.

Figure 5.38: Mesh generated by applying the Marching Cubes algorithm to the output from Fast Marching Segmentation. Notice the middle part of the aorta is fragmented; this was caused by brighter regions surrounding the aorta in the dataset. In this case, stent was around the aneurysm.

Figure 5.39: Mesh generated by applying the Marching Cubes algorithm to the output from Fast Marching Segmentation. To avoid fragmentation, a threshold was applied to this dataset to remove the brighter regions.

The time required to segment the two datasets was recorded for comparison. Figure 5.40 shows the resulting times to perform the three main steps on each part of the datasets. The average total time was less than 8 minutes. The time required for applying the Hessian Vesselness filter varies from almost 3 minutes to 4.5 minutes, depending on the number of iterations performed. Fast Marching segmentation was able to very quickly segment the vessels from the vesselness values due to the high gradients involved. Most of the time is spent placing the seeds and previewing the results to ensure the entire structure is extracted. In total, the STM001 dataset was segmented in 27 minutes and STM007 was segmented in 20 minutes. To give an idea of the time for other techniques:

- A very precise manual segmentation of these datasets took 928 minutes and 450 minutes respectively [85].
- A precise 2.5D hierarchical segmentation of these datasets took 122 minutes and 77 minutes respectively [85].

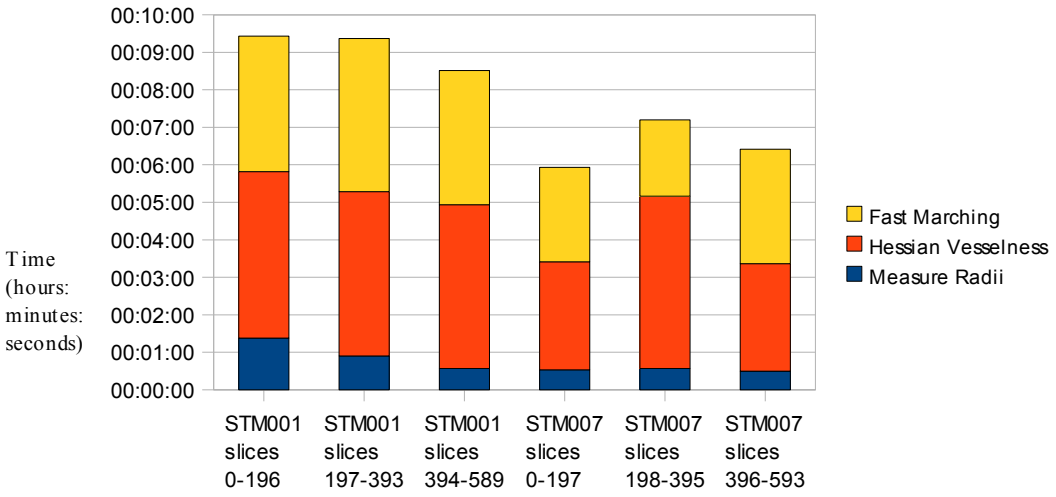


Figure 5.40: Times required to segment the vessels in two datasets. Each dataset was split into three equal parts. The total time is divided into three parts for the different steps: measuring the radii of vessels in the image, calculating the Vesselness of each voxel, and performing Fast Marching segmentation to extract a single connected structure.

The Hausdorff distances between the resulting segmentations and corresponding manual segmentations were calculated with MESH [84]. Figure 5.41 shows the minimum, maximum, mean and RMS distances for both datasets. The distances for the STM001 dataset are significantly higher: the mean distance, 774.78 mm for STM001, is almost 50 times as large as for the STM007 dataset, with 10.97 mm. This difference is caused partially by the fragmentation around the aneurysm, but the large number of small vessels around the aorta arch has the most impact, as seen in Figure 5.43. The segmentation of the STM007 dataset has less fragmentation because of the applied upper threshold and less small vessels protruding from the aorta, so its Hausdorff distances are much smaller.

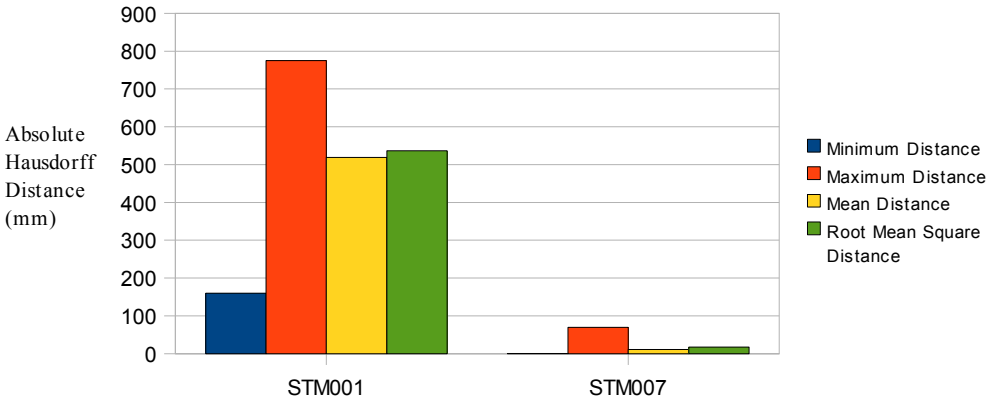


Figure 5.41: Comparison of the Hausdorff distances for the two datasets. The distances are significantly lower for the STM007 dataset because less extra vessels around the aorta arch were segmented.

The minimum distance for the STM001 dataset was a very high 159.8 mm, while the maximum distance was even larger at 774.78 mm. The mean distance and root mean square distance were 519.14 mm and 536.83 mm, respectively. For comparison, the mean distance of the Hierarchical Segmentation was 2.37 mm. A threshold should be applied to help reduce the fragmentation, and the Hessian Vesselness should be reapplied with a larger minimum radius to reduce the number of small vessels around the aorta arch. These changes would produce results more similar to those from the STM007 dataset.

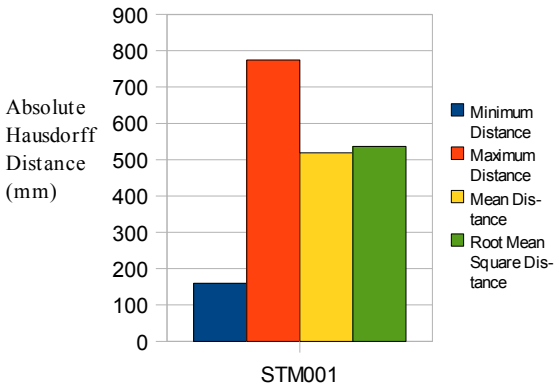


Figure 5.42: Hausdorff distances for the STM001 dataset.

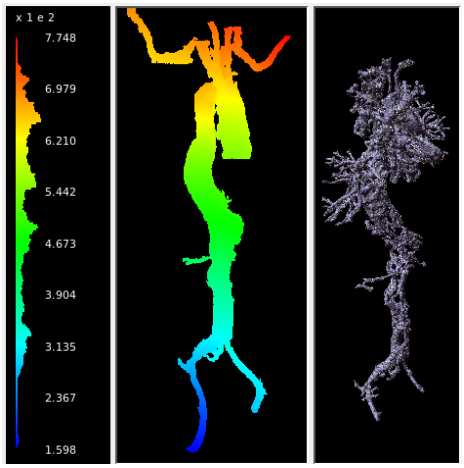


Figure 5.43: Comparison of the results to a manual segmentation of dataset STM001. Darker colours (blue) indicate low Hausdorff distances, while lighter colours (green, yellow, and red) indicate larger Hausdorff distances.

The minimum distance for the STM007 dataset was a low 0.06 mm, while the maximum distance was 69.08 mm. The mean distance and root mean square distance were 10.97 mm and 17.07 mm, respectively. These results are much more accurate than those from the STM001 dataset, comparing better with the Hierarchical Segmentation's mean distance of 2.37 mm. The user would still need to change the radius values to remove some small vessels to better match the results. More seeds could also be placed to capture the ends of the vessels which had become fragmented in the vesselness segmentation.

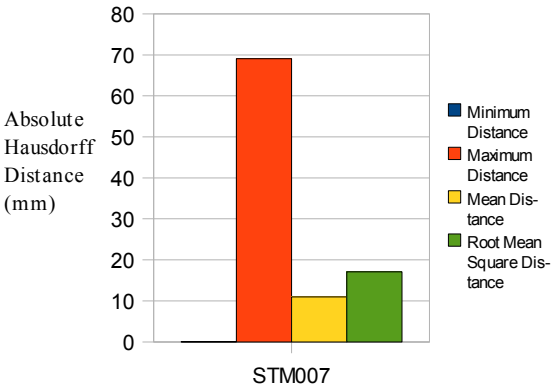


Figure 5.44: Hausdorff distances for the STM007 dataset.

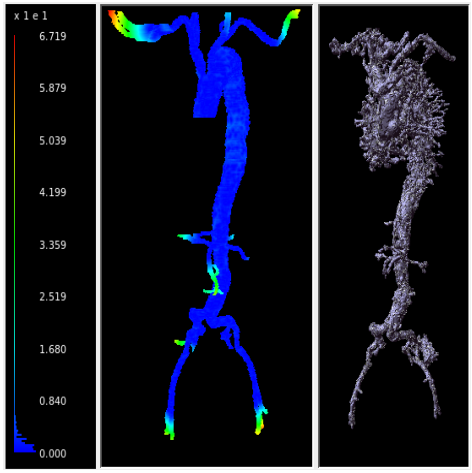


Figure 5.45: Comparison of the results to a manual segmentation of dataset STM007.

From this experiment, it is apparent that the Hessian Vesselness filter can provide a quick segmentation of vessel structures. Even though the results are not as detailed as the previously presented methods, it gives a good approximation of the contour of the vessels in a very short time for such large datasets. The user has the power to select the size of the desired vessels, allowing very small vessels to either be extracted or ignored. By applying Fast Marching segmentation to the resulting vesselness, specific structures can be chosen and extracted. Although the user has some control over this process, the Hessian Vesselness filter is mostly automatic and may yield fragmented structures. Reapplying the filter with different values may be required to improve the segmentation, but this will increase the amount of processing time. Other tools could be used to improve the output vesselness by connecting fragments and improving connectivity of the desired vessels as it has been shown with the threshold removing the brighter values. Undesired branches also could be filled in to remove them from the resulting segmentation.

5.4 - Skeletonisation

Two manually segmented patient datasets (STM001 and STM007) were selected to analyse the performance of the skeletonisation algorithms. These datasets contain a the aorta and other connected blood vessels such as the renal and iliac arteries. To reduce the computation required, each of the datasets was first divided into three parts with the Region of Interest Tool in the same fashion as it was done for the Hessian Vesselness tests. The Skeletonisation Tool was used to find a set of skeletal points which provides a compact representation of the segmented regions, as shown for the first part of the STM001 dataset in Figure 5.46. After the points were found, a centreline was generated from them with the Centreline Tool. Figure 5.47 demonstrates the centreline found from the skeletal points shown in Figure 5.46.

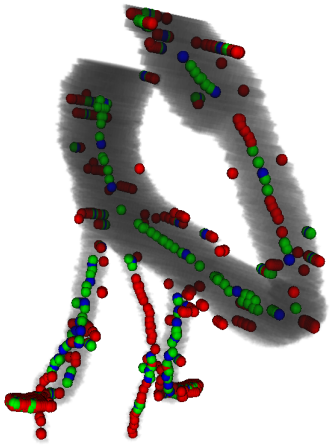


Figure 5.46: Skeletal points found from the one part of the segmented STM001 dataset. The green points are removed during skeletonisation and merged with the blue bifurcation points.

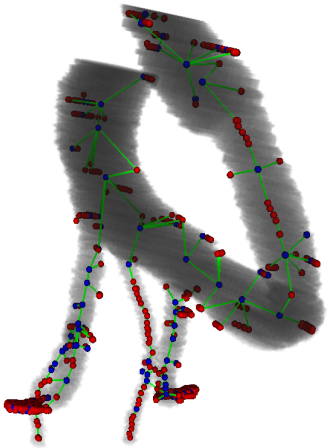


Figure 5.47: Centreline found from the skeletal points. The red points are unchanged, while the blue points are the results from merging points around bifurcations.

The effect of the thinning amount on the skeletonisation process was compared for the first 196 slices of the STM001 dataset. The density of the skeleton is controlled through the thinning amount: a skeletal point is discarded when the difference between its radius and the average of its 26 neighbours' radii is less than the specified amount. A thinness value of 0 will produce the thickest skeleton, while larger values will generate thinner skeletons with less skeletal points. A thinness of 0 generated 976 points, while a thinness of 0.5 generated 629 points. Both thinness values required about 1.5 minutes to perform the skeletonisation. The difference in the number of skeletal points greatly affects the time required to find the centreline: a thinness of 0 required almost 5 minutes, while a thinness of 0.5 required only 1.5 minutes. Because a thinness of 0.5 required about 60% less time than a thinness of 0 and still provided a reasonable set of skeletal points, the thinness of 0.5 was applied while skeletonising the remaining parts.

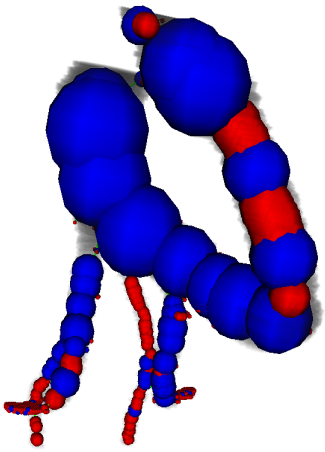


Figure 5.48: Preview of reconstructed vessels by placing a sphere at each skeletal point.

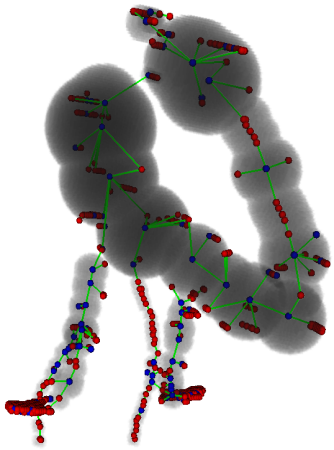


Figure 5.49: Reconstruction of the vessels from the skeletal points.

For each part, the segmented vessels were first skeletonised to find a set of skeletal points, which provide a compact representation of the structure. A thinning amount of 0.5 was used for each skeletonisation to discard points whose radius was within 0.5 of its average neighbour radius. This allowed for a thinner, simpler skeleton with fewer points as it has been shown for the first part of STM001. After generating skeletal points, a centreline was found from these points. The connectedness of the centreline was enforced to ensure a single continuous structure, and points around bifurcations were merged to improve the resulting line. A merge amount of 0.5 was used to merge only points whose radius overlapped a bifurcation point's radius by 50%. This amount provides a fair balance between the loss of detail due to merging and the greater complexity of the centreline without merging. The amount of merging also affects the reconstruction, since more merging will create larger spheres, as shown for the first part of the STM001 dataset in Figure 5.48 and 5.49.

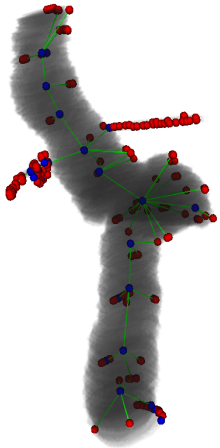


Figure 5.50: Skeletal points and centreline found for the second part of the STM001 dataset.

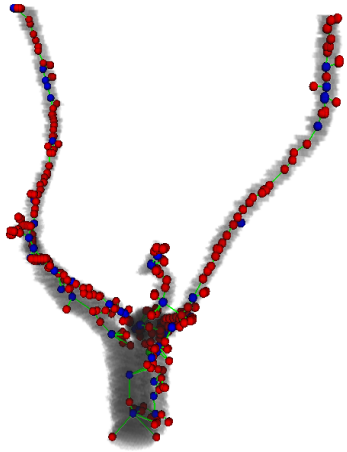


Figure 5.51: Skeletal points and centreline found for the third part of the STM001 dataset.

The resulting centrelines and skeletal points for each part of the STM001 dataset are shown in Figures 5.47, 5.50, and 5.51. The centrelines for the STM007 dataset are shown in Figures 5.52, 5.53, and 5.54. The times required to generate the skeletal points and to find a centreline were also recorded for comparison, as shown in Figure 5.55.

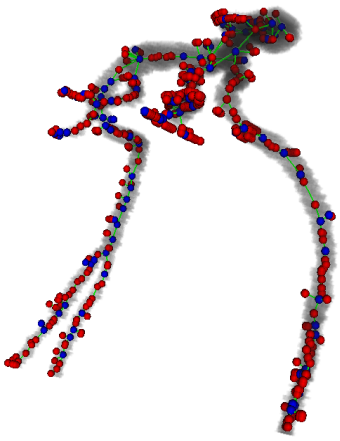


Figure 5.52: Skeletal points and centreline found for the first part of the STM007 dataset.

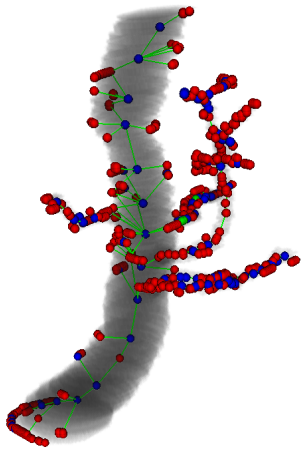


Figure 5.53: Skeletal points and centreline found for the second part of the STM007 dataset.

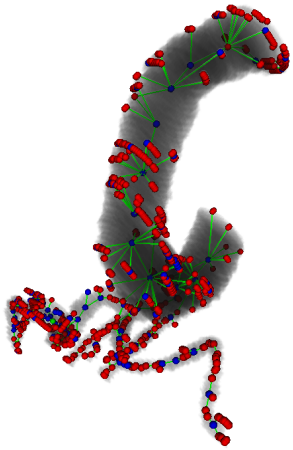


Figure 5.54: Skeletal points and centreline found for the third part of the STM007 dataset.

The average time to generate the skeletal points was slightly more than 1 minute. This time varies between 20 seconds to about 2.5 minutes. Slices 393-589 of the STM001 dataset contain mostly thinner vessels, so the computation time to find the radii of each point was much less, greatly reducing the overall time. In contrast, slices 395-593 of the STM007 dataset contain much of the larger diameter aorta, increasing the time to compute each radius.

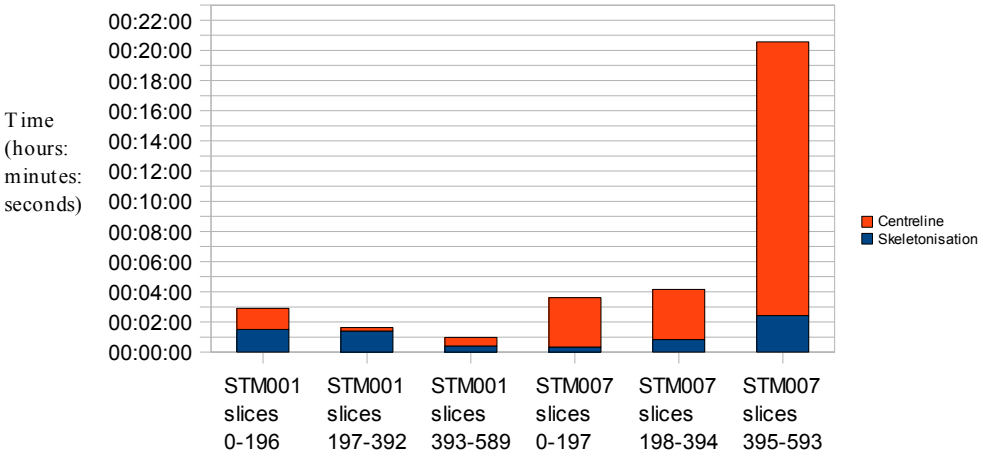


Figure 5.55: Times required to skeletonise the two segmented datasets. Each dataset was split into three equal parts. The total time is divided into two steps: finding a compact set of skeletal points and finding a representative centreline passing through the points.

The amount of time required to find the centreline depends directly on the number of skeletal points found during skeletonisation. The computation time for the centreline is $O(n^2)$ due to the minimum spanning tree algorithm involved. Thus, the time increases quickly as the number of points increases. The number of skeletal points generated for each dataset part was also recorded for comparison. As seen in Figure 5.56, the STM001 dataset has a significantly lower number of skeletal points than STM007: 1392 skeletal points compared to 2926. The STM007 dataset has a greater number of thin vessels (apparent in Figure 5.60 compared to Figure 5.57), causing the skeletal points to be more closely placed. The effect of the number of points is apparent in Figure 5.55, with centreline computation times for STM007 being significantly higher than for STM001. Figure 5.56 also compares the original number of skeletal points to the number of skeletal points after generating the centreline. While creating the centreline, skeletal points near bifurcations were merged together, reducing the number of points and complexity of the centreline. The total number of skeletal points for the STM001 dataset drops by 401 to 991, and the number for the STM007 dataset drops by 878 to 2048.

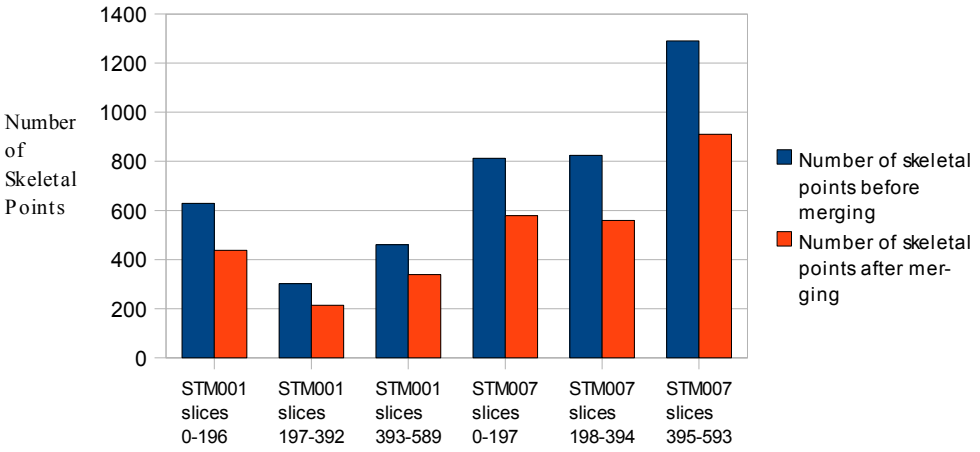


Figure 5.56: Number of skeletal points found after skeletonisation of each dataset part, and the number of remaining skeletal points after merging around bifurcations in the centreline.

After skeletonising the three parts of both datasets, the parts were combined to produce the full skeletonisation of the complete dataset, as shown in Figures 5.57 and 5.60. The datasets were also reconstructed from the radii provided by each skeletal point. A preview of the reconstruction accomplished by placing a sphere at each skeletal point is shown in Figures 5.58 and 5.61. The spheres were then filled to reconstruct the original vessels, as shown in Figures 5.59 and 5.62.

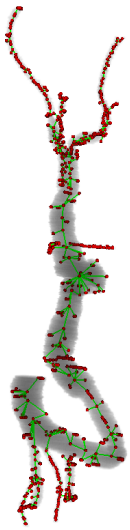


Figure 5.57: Full centreline for the complete STM001 dataset.

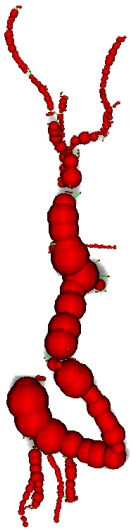


Figure 5.58: Preview reconstruction of the STM001 dataset.

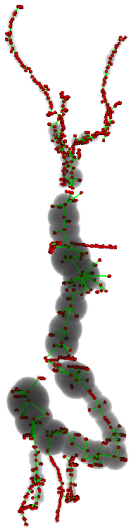


Figure 5.59: Reconstruction of the STM001 dataset

From these figures, it can be scene that the reconstruction is only an approximation. The spheres used to reconstruct the vessels are still very noticeable. Although the merging of points around bifurcations improves the centreline, the resulting skeletal points provide a less accurate approximation. The resulting points tend to have larger spheres because they must encompass the spheres of all the merged points. These spheres will often protrude from the original vessel structure, making it more round or bumpy in places where it should be flat or smooth. This problem could be addressed by smoothing the surface of the resulting mesh either with a smoothing filter or by applying a closing operation (morphological operation performing first a dilation and then an erosion). Because each dataset was split into three parts before skeletonisation, the point where one part ends and the next part begins is visible in the complete skeletonisation and reconstruction. Many skeletal points tend to be generated where a vessel crosses the border of the image, reducing the quality of the centreline. These points also have small spheres which cause the reconstructed vessels to be fragmented. If the dataset was treated as a whole instead of splitting it, this problem would disappear. Nevertheless, a fairly new computer would be needed to deal with the large amount of data generated for such datasets.

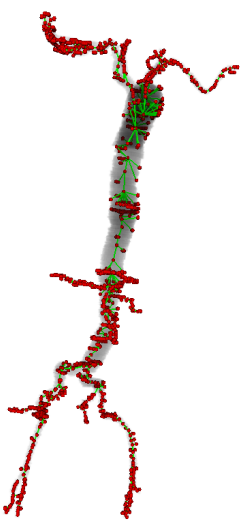


Figure 5.60: Full centreline for the complete STM007 dataset.

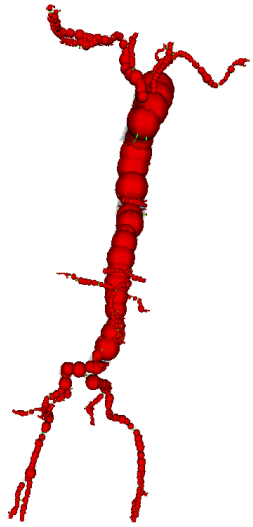


Figure 5.61: Preview reconstruction of the STM007 dataset.

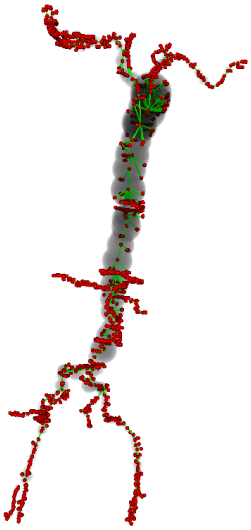


Figure 5.62: Reconstruction of the STM007 dataset.

The Hausdorff distances between the resulting reconstructed vessels and corresponding manual segmentations were calculated with MESH [84]. Figure 5.63 shows the minimum, maximum, mean and RMS distances for both datasets. The distances are very similar, with the maximum distance differing the most, by 2.88 mm. The minimum distances are 0 mm for both datasets. The mean distances are 1.77 mm for STM001 and 1.61 mm for STM007, proving that the skeletonisation gives segmented results very close to the manual segmentation. The root mean square distances are 2.38 mm for STM001 and 2.26 mm for STM007.

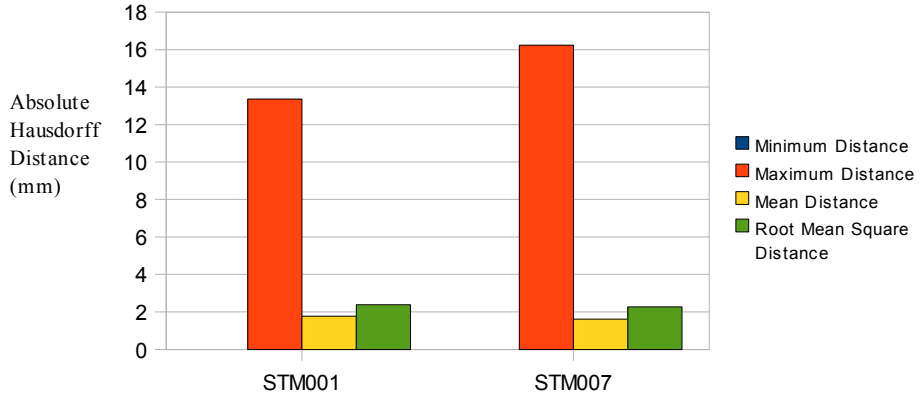


Figure 5.63: Comparison of the Hausdorff distances between the reconstructions and original segmented vessels for the two datasets. The blue bars are not visible because the minimum distances are 0 mm.

Demonstrated in Figures 5.64 and 5.65, the most difference between the reconstruction and the original vessels occurs at the end of the aorta. In the reconstruction, the end is a large sphere, whereas the original end is more like a cylinder. The large spheres created during the merging of points around bifurcations also increases the distance to about 5 mm in areas displayed as green in the figures. Because the distances are small enough to provide a simple reconstruction of the original vessels, the skeletal points and centreline provide a good compact representation of the vessels.

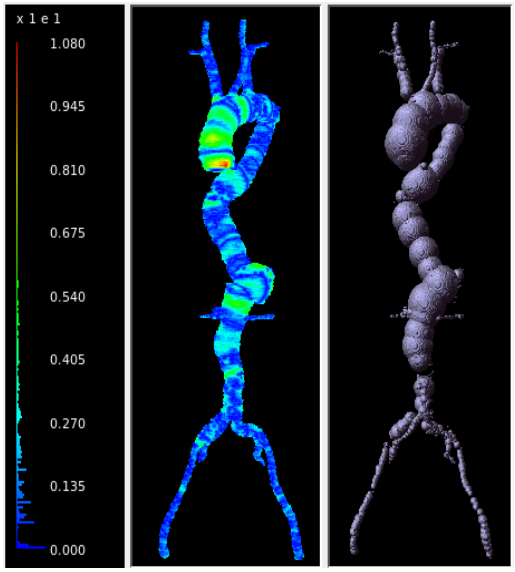


Figure 5.64: MESH comparison of the reconstruction to the manual segmentation of dataset STM001.

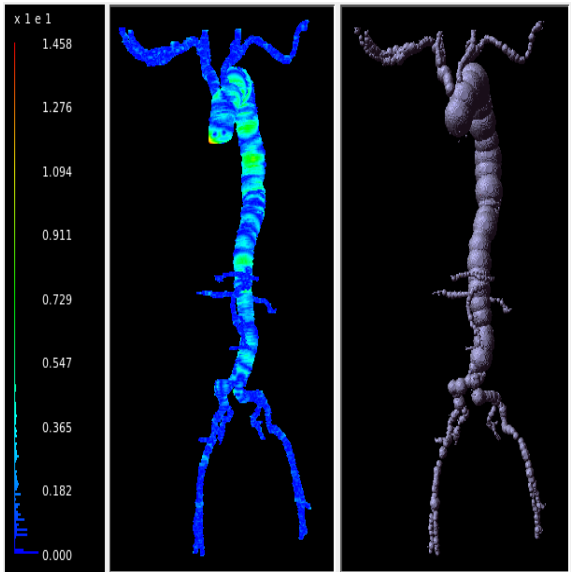


Figure 5.65: MESH comparison of the reconstruction to the manual segmentation of dataset STM007.

6 - Conclusions

As seen from the experimentation, the implemented tools and algorithms provide promising results. Hierarchical segmentation was found to be a good alternative to level set methods, and the combination of the Hessian Vesselness filter with Fast Marching Segmentation yields a quick method of extracting even small vessels. The skeletonisation algorithms were shown to provide a reasonable compact representation of vascular structures from which information about the vessels can be derived. Thanks to an intuitive user interface, the application allows the user to easily choose from sets of tools to accomplish a variety of goals. By combining these tools together in a sequence, an anatomical model can be generated from a patient dataset.

6.1 - Applications

The program is currently being applied to both vascular and non-vascular projects. Although the segmentation methods have not found many applications yet, the mesh generation functionality alone has proved very helpful and easy to use.

Rafal Blazewski has developed a virtual catheterisation simulator that allows a user to practice catheterisation procedures. The vascular meshes were generated using the program's meshing functionality. Figure 6.1 shows the aorta mesh that was generated with the Surface Mesher Tool from a manually segmented dataset. The catheter, which can be guided by the user, is also shown. Figure 6.2 shows not only the vessels and catheter, but also a simulated X-ray image that would be available to a surgeon during catheterisation.

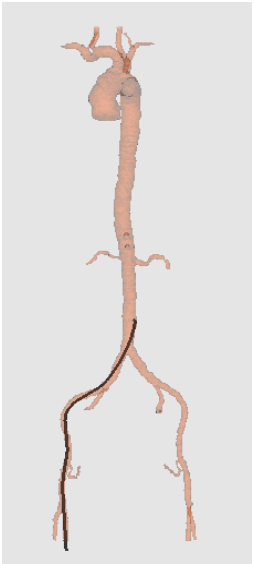


Figure 6.1: View of the vascular structure (red) and the inserted catheter (black).

Figure 6.2: View of the vascular structure (red) and the inserted catheter (black). The skeleton is also visible here as a volumetric rendering to simulate an X-ray image.

Dr. Pierre-Frederic Villard's work on breathing simulations has also made use of the program for mesh generation. The Surface Mesher Tool was used to create a triangular surface mesh of the diaphragm from a segmented dataset. Figure 6.3 shows the mesh generated with a mesh angle of 5, mesh distance of 1, and mesh radius of 1. The mesh radius in this case had to be very small because of the thinness of the diaphragm. The small radius leads to a very high resolution mesh, with many small bumps. These bumps were smoothed out by applying 200 iterations of Windowed Sinc smoothing with a pass band of 0.2. The resulting mesh was then decimated to reduce the resolution by 50%. Another 200 iterations of Windowed Sinc smoothing were applied, again with a pass band of 0.2. Figure 6.4 shows the mesh that results from this process. The resulting diaphragm mesh was then used in the breathing simulation displayed in Figure 6.5.

Figure 6.3: Mesh generated via CGAL's algorithms with the Surface Mesher Tool. A mesh angle of 5, mesh distance of 1, and mesh radius of 1 were used to generate the entire surface of the diaphragm.

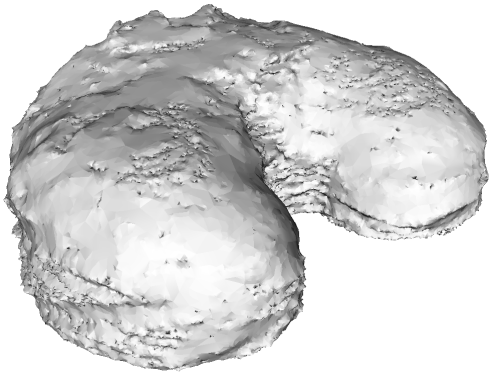


Figure 6.4: Resulting mesh after 200 iterations of Windowed Sinc smoothing with a pass band of 0.2, followed by a 50% decimation, followed by another 200 iterations of Windowed Sinc smoothing.

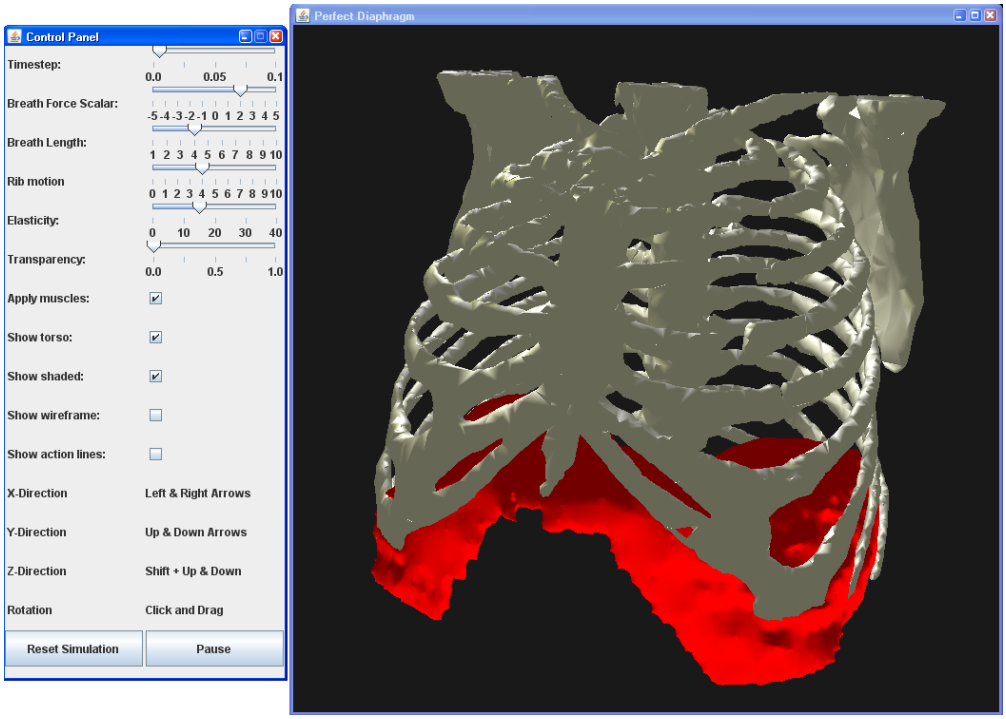


Figure 6.5: Diaphragm breathing simulation using the generated diaphragm mesh.

The mesh generation tools were also used in the CRaIVE project. Figure 6.6 shows meshes of several organs and skeletal structures that were generated using the Marching Cubes or Surface Mesher Tools. Meshes for deformable objects were created by tetrahedralising the surface mesh. Alejandro Granados combined these tetrahedra with his mass-spring model to allow deformations and return haptic feedback to the user. Figure 6.7 shows an example of his interactive simulation of a liver.

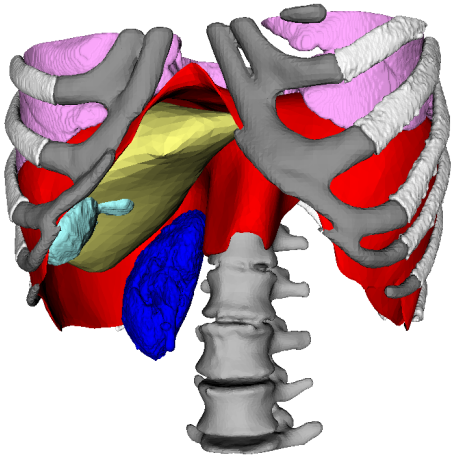


Figure 6.6: Organs and skeleton of a patient meshed with the Marching Cubes or Surface Mesher Tools.

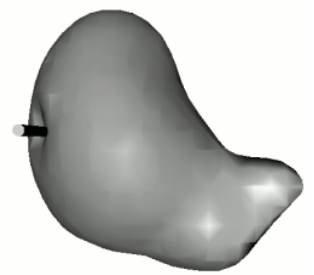


Figure 6.7: Interactive simulation of a deformable liver. The surface of the liver was generated with the Surface Mesher Tool, and the tetrahedral mesh was generated with the Tetrahedralisation Tool.

6.2 - Future

The current implementation allows more tools to be easily integrated to provide further functionality. Alternative or improved segmentation, skeletonisation, and mesh generation algorithms will be added in the future. Continued experimentation of the current methods may also yield better results.

The segmentation methods should be further tested to determine how effective and robust they are in a variety of situations. The 2.5D Hierarchical Segmentation may prove useful for high contrast structures other than blood vessels. Much more experimentation of the Hessian Vesselness filter is needed to determine if the many extra small vessels can be ignored or filtered out. If these vessels can be removed from the vesselness image, then the Fast Marching Segmentation will be able to produce a result very similar to manual segmentation but in less time. Further segmentation algorithms can also be added to provide alternative methods for different types of structures. However, the focus of the program will remain around vascular segmentation.

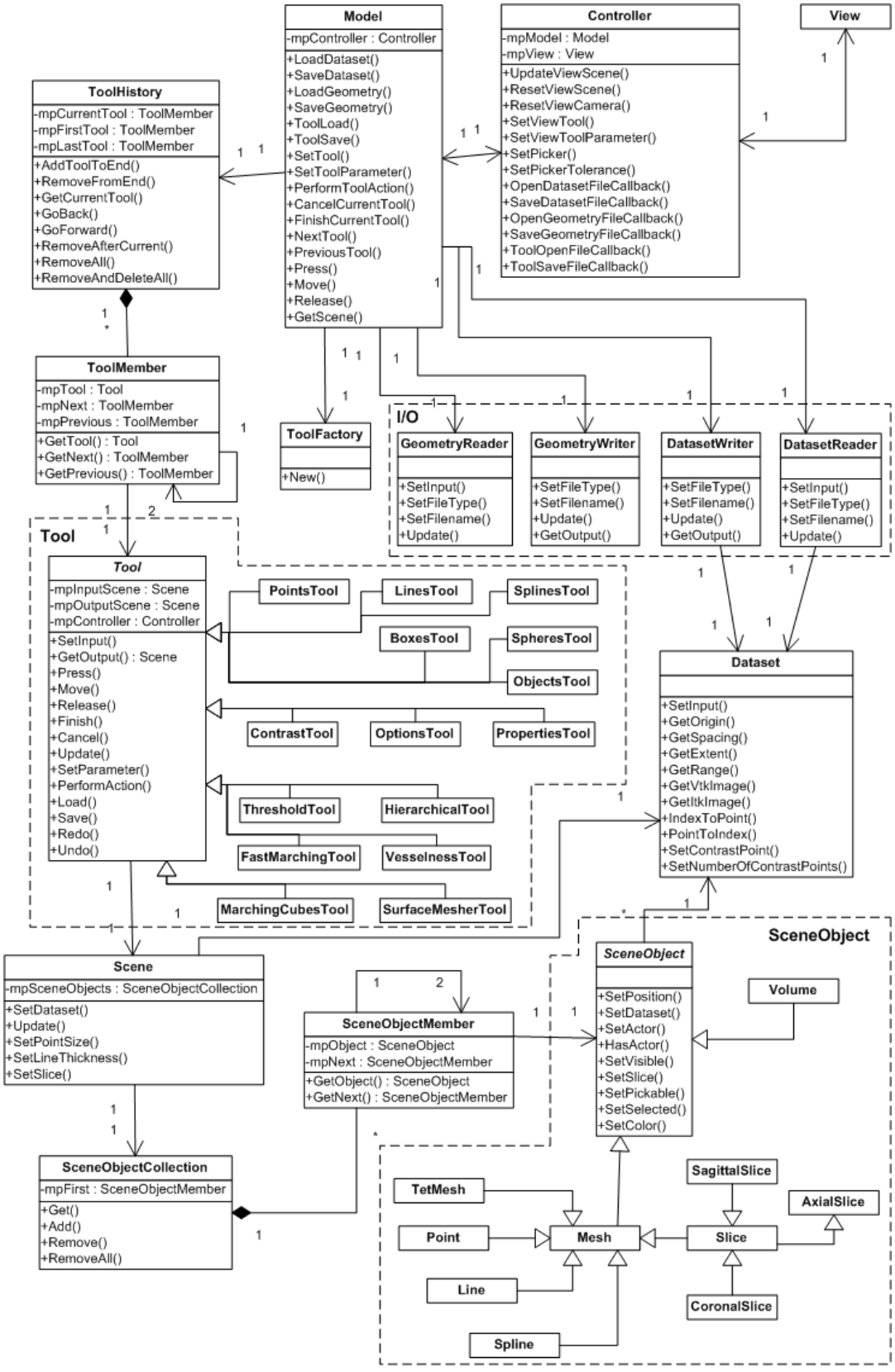
The skeletonisation algorithms could also be further developed in several ways. Optimisation of the algorithms can decrease the computation time to find both the skeletal points and centreline. Different methods may help reduce the number of branches caused by noise on the surface of the segmented vessels. More work on the reconstruction from skeletal points may help to better approximate the original vessels. Simple smoothing of the reconstruction can merge the individual spheres to provide a smooth surface that may be more accurate. Editing tools to allow modification of the centreline will be needed to allow the user to merge and split branches. An improved centreline will provide a better representation of the original vessels.

The mesh generation algorithms are quite comprehensive already; however, an edge contraction algorithm which does not require a manifold mesh would be a very useful alternative for simplifying meshes. More mesh editing tools could also allow the user to manually improve the resulting meshes.

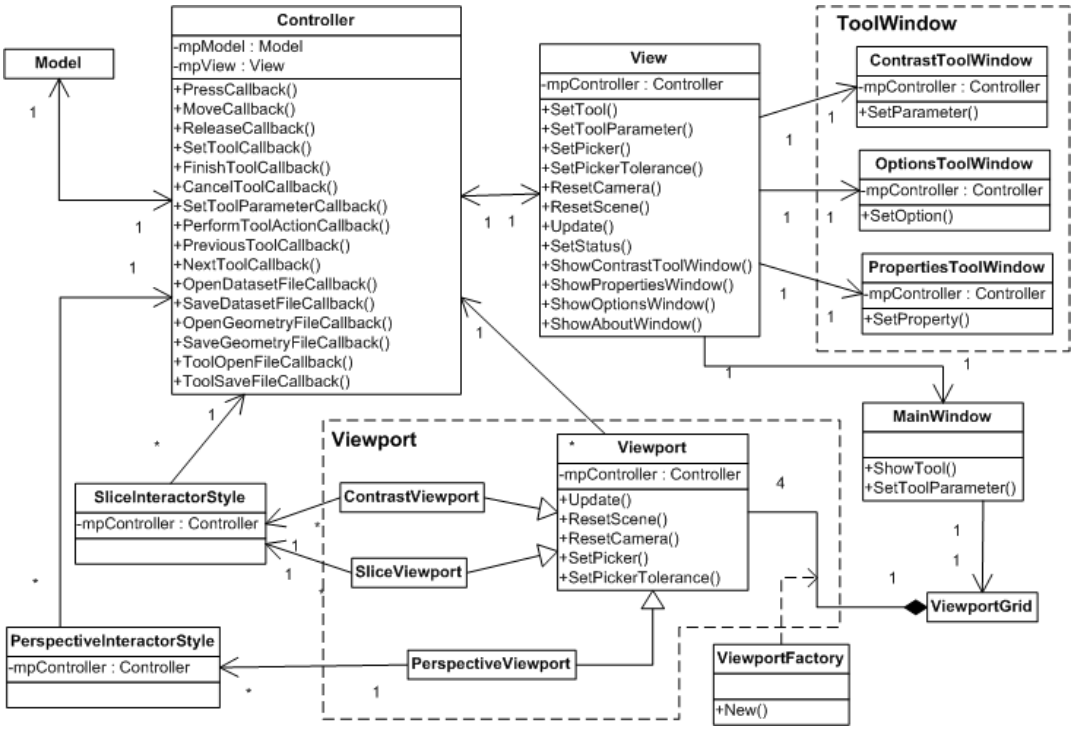
Overall, more optimisation in both memory usage and computation could greatly improve the program. The large datasets involved in medical imaging push a standard computer to the limits, especially when some algorithms require several copies at the same time. Many of the algorithms do not take advantage of multiple threads which could greatly reduce computation times and allow for more interactive methods. More experimentation with the current algorithms will likely lead to interesting results, which can be used to further develop the tools. Therefore, this project has great potential for future improvements and extensions.

7 - Appendix

7.1 - UML Class Diagram of the Model Component



7.2 - UML Class Diagram of the View Component



8 - Bibliography

[1] M. Xin, Z. Lei, I. Volkau, Z. Weili, A. Aziz, M. H. Ang, Jr., and W. L. Nowinski,"A Virtual Reality Simulator for Remote Interventional Radiology: Concept and prototype design," *IEEE Transactions on Biomedical Engineering*, vol. 53, no. 8, pp. 1696-1700, August 2006.

[2] F. P. Vidal, F. Bello, K. W. Brodlie, N. W. John, D. Gould, R. Phillips, and N. J. Avis,"Principles and Applications of Computer Graphics in Medicine," *Computer Graphics Forum*, vol. 25, no. 1, pp. 113-137, March 2006.

[3] T. Alderlienstent, "Simulation of Minimally-Invasive Vascular Interventions for Training Purposes", Ph.D. dissertation, Utrecht Graduate School for Biomedical Image Sciences, Utrecht, The Netherlands, 2004.

[4] J. Dankelman, M. Wentink, C. A. Grimbergen, H. G. Stassen, and J. Reekers,"Does Virtual Reality Training Make Sense in Interventional Radiology? Training skill-, rule-, and knowledge-based behavior," *CardioVascular and Interventional Radiology*, vol. 27, no. 5, pp. 417-421, September 2004.

[5] S. Johnson, A. Healy, J. Evans, M. Murphy, M. Crawshaw, and D. Gould,"Physical and Cognitive Task Analysis in Interventional Radiology," *Clinical Radiology*, vol. 61, no. 1, pp. 97-103, January 2006.

[6] N. E. Seymour, A. G. Gallagher, S. A. Roman, M. K. O'Brien, V. K. Bansal, D. K. Anderson, and R. M. Satava,"Virtual Reality Training Improves Operating Room Performance: Results of a randomized, double-blinded study," *Annals of Surgery*, vol. 236, no. 4, pp. 458-463, October 2002.

[7] N. W. John and Ik Soo Lim,"Cybermedicine Tools for Communication and Learning," *Journal of Visual Communication in Medicine*, vol. 30, no. 1, pp. 4-9, March 2007.

[8] A. Zorcolo, E. Gobbetti, G. Zanetti, and M. Tuveri,"A Volumetric Virtual Environment for Catheter Insertion Simulation," *6th Eurographics Workshop on Virtual Environments*, vol. 6, no. 1, pp. 57-64, June 2000.

[9] N. W. John and R. F. McCloy,"Navigating and Visualizing Three-dimensional Data Sets," *The British Journal of Radiology*, vol. 77, no. 918, pp. S108-S113, June 2004.

[10] L. D. Griffen, A. C. F. Colchester, S. A. Roll, and C. S. Studholme,"Hierarchical Segmentation Satisfying Constraints," *Proceedings of the British Machine Vision Conference*, vol. 5, no. 1, pp. 135-144, September 1994.

[11] N. Gagvani and S. Deborah, "Parameter Controlled Skeletonization of Three Dimensional Objects", Rutgers State University, New Jersey, USA, CAIP-TR-216, 1997.

[13] H. Si and K. Gartner,"Meshing Piecewise Linear Complexes by Constrained Delaunay Tetrahedralizations," *Proceedings of the 14th International Meshing Roundtable*, vol. 14, no. 1, pp. 147-164, September 2005.

[14] Bill Spitzak et al., "Fast Light Toolkit (FLTK)," *FTLK: Fast light toolkit*. [Online]. Available: http://www.fltk.org/. [Accessed: 11 March 2008].

[15] Kitware, Inc., "Insight Segmentation and Registration Toolkit," *NLM Insight Segmentation & Registration Toolkit*. [Online]. Available: http://www.itk.org/. [Accessed: 11 March 2008].

[16] Kitware, Inc., "VTK Home Page," *The Visualization Toolkit*. [Online]. Available: http://www.vtk.org/. [Accessed: 11 March 2008].

[17] Max Planck Institute for Computer Science et al., "CGAL - Computational Geometry Algorithms Library," *CGAL*. [Online]. Available: http://www.cgal.org/. [Accessed: 11 March 2008].

[18] H. Si, "TetGen: A Quality Tetrahedral Mesh Generator," *TetGen: A Quality Mesh Generator and Three-Dimensional Delaunay Triangulator.* [Online]. Available: http://tetgen.berlios.de/. [Accessed: 11 March 2008].

[19] Kitware, Inc., "CMake Cross Platform Make," *CMake*. [Online]. Available: http://www.cmake.org/. [Accessed: 11 March 2008].

[20] A. Szekely, "Nullsoft Scriptable Install System," *NSIS*. [Online]. Available:

http://nsis.sourceforge.net/. [Accessed: 20 July 2008].

[21] Dimitri van Heesch, "Doxygen: Source code documentation generator tool," *Doxygen*. [Online]. Available: http://www.stack.nl/~dimitri/doxygen/. [Accessed: 15 June 2008].

[22] U. Kuhnapfel, H. K. Cakmak, and H. MaaB,"Endoscopic Surgery Training using Virtual Reality and Deformable Tissue Simulation," *Computers and Graphics*, vol. 24, no. 1, pp. 671-682, February 2000.

[23] G. D. Picetti, "Basic Introduction to Minimally Invasive Spine Surgery," *SpineUniverse*. [Online]. Available: http://www.spineuniverse.com/displayarticle.php/article240.html. [Accessed: 27 March 2008].

[24] Society of Interventional Radiology, "Stroke Prevention & Treatment," *Society of Interventional Radiology*. [Online]. Available: http://www.sirweb.org/patPub/strokeTreatments.shtml. [Accessed: 28 March 2008].

[25] University of Virginia, "CV Catheterization," *University of Virginia Health System*. [Online]. Available: http://www.healthsystem.virginia.edu/internet/anesthesiology-elective/cardiac/cvcath.cfm. [Accessed: 28 March 2008].

[26] Rhode Island Vascular Insitute, "Kidney Artery Disease," *Rhode Island Medical Imaging*. [Online]. Available: http://www.rivascularinstitute.com/medical_conditions/renovascular_conditions.html. [Accessed: 28 March 2008].

[27] Associated Radiologists of Flint, P.C, "Interventional Radiology," *Associated Radiologists of Flint, P.C*. [Online]. Available: http://www.arfpc.com/interventional_radiology.html. [Accessed: 28 March 2008].

[28] H. K. Cakmak and U. Kuhnapfel,"Animation and Simulation Techniques for VR-Training Systems in Endoscopic Surgery," *Eurographics Workshop on Animation and Simulation*, vol. 11, no. 1, pp. 173-185, August 2000.

[29] P. J. Gorman, A. H. Meier, and T. M. Krummel,"Computer Assisted Training and Learning in Surgery," *Computer Aided Surgery*, vol. 5, no. 2, pp. 120-130, June 2000.

[30] F. Peugnet, P. Dubois, and J. F. Rouland,"Virtual Reality Versus Conventional Training in Retinal Photocoagulation: A first clinical assessment," *Computer Aided Surgery*, vol. 3, no. 1, pp. 20-26, December 1998.

[31] K. Moorthy, M. Mansoori, F. Bello, J. Hance, S. Undre, Y. Munz et al.,"Evaluation of the Benefit of VT Simulation in a Multi-media Web-based Educational Tool," *Medicine Meets Virtual Reality*, vol. 12, no. 1, pp. 247-252, January .

[32] D. Meglan,"Making Surgical Simulation Real," *Computer Graphics*, vol. 30, no. 4, pp. 37-39, November 1996.

[33] L. Barker,"CathSim," *Studies in Health Technology and Information*, vol. 62, no. 1, pp. 36-37, January 1999.

[34] S. Cotin, S. Dawson, D. Meglan, D. Shaffer, M. Ferrell, R. Bardsley, F. Morgan, T. Nagano, J. Nikom, P. Sherman, M. Walterman, and J. Wendlandt,"ICTS, an interventional cardiology training system," *Studies in Health Technology and Infromatics*, vol. 70, no. 1, pp. 59-65, January 2000.

[35] S. Cotin, V. Luboz, V. Perforaro, P. E. Neuman, X. Wu, and S. Dawson,"High-fidelity simulation of interventional neuroradiology procedures," *Proceedings of the 40th Annual Meeting of the American Society of Neuroradiology*, vol. 40, no. 1, pp. 443, May 2005.

[36] P. Yushkevich, "ITK-SNAP," *itk-SNAP*. [Online]. Available: http://www.itksnap.org/. [Accessed: 12 March 2008].

[37] H. K. Cakmak and U. Kuhnapfel,"Animation and Simulation Techniques for VR-Training Systems in Endoscopic Surgery," *Eurographics Workshop on Animation and Simulation*, vol. 11, no. 1, pp. 173-185, August 2000.

[38] Harvard University, "IIC/Astromed," *The Astronomical Medicine Project*. [Online]. Available: http://astromed.iic.harvard.edu/. [Accessed: 13 March 2008].

[39] P. A. Yushkevish, J. Piven, H. C. Hazlett, R. G. Smith, S. Ho, J. C. Gee, and G. Gerig,"User-guided 3D Active Contour Segmentation of Anatomical Structures: significantly improved efficiency and

reliability," *NeuroImage*, vol. 31, no. 1, pp. 1116-1128, March 2006.

[40] H. Sun, Y. Zhao, F. Hong, M. Li, and J. Mei,"A System for Osteosarcoma Segmentation and 3-D Reconstruction," *International Conference on Communications, Circuits and Systems 2004*, vol. 2, no. 1, pp. 950-954, June 2004.

[41] G-Z Yang, "Region Based Segmentation," *Imperial College: Computer Vision (c4.18) Lecture Notes*. [Online]. Available: http://www.doc.ic.ac.uk/~gzy/teaching/vision/vision-s04.pdf. [Accessed: 27 March 2008].

[42] R. Adams and L. Bischof,"Seeded Region Growing," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 16, no. 6, pp. 641-647, June 1994.

[43] R. Beare,"Regularized Seeded Region Growing," *Proceedings of the Sixth International Symposium for Mathematics Morphology*, vol. 6, no. 1, pp. 91-99, April 2002.

[44] T. Kapur, P. A. Beardsley, S. F. Gibson, W. E. L. Grimson, and W. M. Wells,"Model Based Segmentation of Clinical Knee MRI," *Proceedings of IEEE International Workshop on Model-Based 3D Image Analysis*, vol. 1, no. 1, pp. 97-106, January 1998.

[45] F. Meyer and S. Beucher,"Morphological Segmentation," *Journal of Visual Communication and Image Representation*, vol. 1, no. 1, pp. 21-46, September 1990.

[46] M. Wasilewski,"Active Contours using Level Sets for Medical Image Segmentation," *Proceedings of the 45th Annual Southeast Regional Conference*, vol. 45, no. 1, pp. 19-23, March 2007.

[47] R. Malladi, J. A. Sethian, and B. C. Vemuri,"Shape Modeling with Front Propagation: a level set approach," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 2, pp. 158-175, February 1995.

[48] T. F. Chan and L. A. Vese,"Active Contours Without Edges," *IEEE Transactions on Image Processing*, vol. 10, no. 1, pp. 266-277, February 2001.

[49] B. Sumengen and B. S. Manjunath,"Graph Partitioning Active Contours (GPAC) for Image Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 4, pp. 509-521, April 2006.

[50] M. Lynch, O. Ghita, and P. F. Whelan,"Segmentation of the Left Ventricle of the Heart in 3-D+t MRI Data Using and Optimized Nonrigid Temporal Model," *IEEE Transactions on Medical Imaging*, vol. 27, no. 2, pp. 195-203, February 2008.

[51] N. Gagvani and S. Deborah, "Parameter Controlled Skeletonization of Three Dimensional Objects", Rutgers State University, New Jersey, USA, CAIP-TR-216, 1997.

[52] T. Pavlidis,"A Thinning Algorithm for Discrete Binary Images," *Computer Graphics and Image Processing*, vol. 13, no. 1, pp. 142-157, May 1980.

[53] C. Arcelli and G. Sanniti di Baja,"A Width-Independent Fast Thinning Algorithm," *IEEE Transactions on Pattern Recognition and Machine Intelligence*, vol. 7, no. 4, pp. 463-474, March 1985.

[54] R. J. T. Sadleir and P. F. Whelan,"Fast Color Centerline Calculation Using Optimised 3D Topological Thinning," *Computerized Medical Imaging and Graphics*, vol. 29, no. 6, pp. 251-258, September 2004.

[55] R. Singh, V. Cherkassky, and N. Papanikolopoulos,"Self-Organizing Maps for the Skeletonization of Sparse Shapes," *IEEE Transactions on Neural Networks*, vol. 11, no. 1, pp. 241-248, September 2000.

[56] C. Wang and O. Smedby,"Coronary Artery Segmentation and Skeletonization Based on Competing Fuzzy Connectedness Tree," *Medical Imaging and Computer-Assisted Intervention 2007*, vol. 10, no. 1, pp. 311-318, October 2007.

[57] C. Fouard, G. Malandain, S. Prohaska, M. Westerhoff, F. Cassot, C. Mazel, D. Asselot, and J. P. Marc-Vergnes,"Skeletonization by Blocks for Large 3D Datasets: Application to brain microcirculation," *IEEE International Symposium for Biomedical Imaging*, vol. 1, no. 1, pp. 89-92, April 2004.

[58] H. E. Bennink, H. C. van Assen, G. J. Streekstra, R. ter Wee, J. A. E. Spaan, and B. M. ter Haar Romeny,"A Novel 3D Multi-scale Lineness Filter for Vessel Detection," *Medical Imaging and*

Computer-Assisted Intervention 2007, vol. 10, no. 1, pp. 436-443, October 2007.

[59] Y. Sato, S. Nakajima, H. Atsumi, T. Koller, G. Gerig, S. Yoshida, and R. Kikinis,"3D Multi-scale line filter for segmentation and visualization of curvilinear structures in medical images," *Lecture Notes in Computer Science*, vol. 1205, no. 1, pp. 213-222, April 1997.

[60] O. Kubassova,"Automatic Segmentation of Blood Vessles from Dynamic MRI Datasets," *Medical Imaging and Computer-Assisted Intervention 2007*, vol. 10, no. 1, pp. 593-600, October 2007.

[61] M. Schaap, R. Manniesing, I. Smal, T. van Walsum, A. vad der Lugt, and W. Niessen,"Bayesian Tracking of Tubular Structures and Its Application to Carotid Arteries in CTA," *Medical Imaging and Computer-Assisted Intervention 2007*, vol. 10, no. 1, pp. 562-570, October 2007.

[62] T. Dey, "Delaunay Mesh Generation of Three Dimensional Domains", Ohio State University, Technical report OSU-CISRC-09/07-TR64, 2007.

[63] W. E. Lorensen and H. E. Cline,"Marching Cubes: A high resolution 3-D surface construction algorithm," *Computer Graphics*, vol. 21, no. 4, pp. 163-169, July 1987.

[64] G. Nielson,"On Marching Cubes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 283-297, April 2003.

[65] J. Sharman, "The Marching Cubes Algorithm," *Exaflop*. [Online]. Available: http://www.exaflop.org/docs/marchcubes/. [Accessed: 30 March 2008].

[66] C. Ho, F. Wu, H. Chen, Y. Chuang, and M. Ouhyoung,"Cubical Marching Squares: Adaptive feature preserving surface extraction from volume data," *Eurographics 2005*, vol. 24, no. 3, pp. 537-545, December 2005.

[67] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen,"Decimation of Triangle Meshes," *Computer Graphics*, vol. 16, no. 2, pp. 65-70, July 1992.

[68] M. Knapp, "Mesh Decimation Using VTK", Institute of Computer Graphics and Algorithms, Vienna University of Technology, Austria, Seminar aus Computergraphik, 2002.

[69] P. Lindstrom and G. Turk,"Fast and memory efficient polygonal simplification," *IEEE Visualization*, vol. , no. 1, pp. 279-286, October 1998.

[70] J-D Boissonnat and S. Oudot,"Provably Good Surface Sampling and Approximation," *Eurographics Symposium on Geometry Processing*, vol. 1, no. 1, pp. 9-19, 265, June 2003.

[71] A. Nealen, T. Igarashi, O. Sorkine, and M. Alexa,"Laplacian Mesh Optimization," *Graphite 2006*, vol. 4, no. 1, pp. 381-389, December 2006.

[72] G. Taubin, "Geometric Signal Processing on Polygonal Meshes", IBM T. J. Watson Research Center, Eurographics, 2000.

[73] H. Si and K. Gartner,"Meshing Piecewise Linear Complexes by Constrained Delaunay Tetrahedralizations," *Proceedings of the 14th International Meshing Roundtable*, vol. 14, no. 1, pp. 147-164, September 2005.

[74] H. Si,"On Refinement of Constrained Delaunay Tetrahedralizations," *Proceedings of the 15th International Meshing Roundtable*, vol. 15, no. 1, pp. 509-528, September 2006.

[75] J. Seward, "Valgrind Home," *Valgrind*. [Online]. Available: http://valgrind.org/. [Accessed: 15 June 2008].

[76] Kitware, Inc., "Documentation," *ITK: Insight Toolkit*. [Online]. Available: http://www.itk.org/Doxygen/html/index.html. [Accessed: 11 March 2008].

[77] Kitware, Inc., "VTK Class List," *The Visualization Toolkit*. [Online]. Available: http://www.vtk.org/doc/nightly/html/annotated.html. [Accessed: 11 March 2008].

[78] G. Lawlor, "VTK," *Bioengineering Research*. [Online]. Available: http://www.bioengineering-research.com/vtk. [Accessed: 11 March 2008].

[79] J. A. Sethian, "Level Set Methods and Fast Marching Methods", Ch. 8, Ed.2; Cambridge Press, 1999.

[80] D. Cherition and R. E. Tarjan,"Finding minimum spanning trees," *SIAM Journal of Computing*, vol. 5, no. 1, pp. 724–741, December 1976.

[81] L. Rineau and M. Yvinec, "3D Surface Mesh Generation," *CGAL Manual*. [Online]. Available: http://www.cgal.org/Manual/3.3/doc_html/cgal_manual/Surface_mesher/Chapter_main.html.

[Accessed: 11 March 2008].

[82] F. Cacciola, "Triangulated Surface Mesh Simplification," *CGAL Manual*. [Online]. Available: http://www.cgal.org/Manual/3.3/doc_html/cgal_manual/Surface_mesh_simplification/Chapter_main.html. [Accessed: 11 March 2008].

[83] H. Si, "Features," *TetGen*. [Online]. Available: http://tetgen.berlios.de/features.html. [Accessed: 11 March 2008].

[84] N. Aspert, "M.E.S.H. : Measuring Error between Surfaces using the Hausdorff distance," *MESH*. [Online]. Available: http://mesh.berlios.de/. [Accessed: 9 September 2008].

[85] N. Din, "Generation of 3D models of anatomy for the simulation of endoluminal minimally access therapy", Project report of BSc in Surgery in Anesthesia. Supervisors: F. Bello and V. Luboz. Imperial College, 2008.

[86] L. Tucker, "Liver Segmentation for Soft Tissue Characterisation in the Context of Realistic Surgical Simulation", Project report of BSc in Reproductive and Development Sciences. Supervisors: F. Bello and V. Luboz. Imperial College, 2008.