Structural Identification in Medical Imaging (SIMI) Final Report

Adrian Clay Daniel Breed Raluca Bolovan Shaan Malik
Savvas Kleanthous

January 9, 2012

Contents

Chapter 1

Introduction

The medical profession produces and uses CAT and MRI scans to non-invasively inspect the body and diagnose problems, without recourse to exploratory surgery.
Segmentation - the process of partitioning an image into segments of interest - is an area of Computer Vision that has application in the medical setting, as if an organ or some other anatomical feature can be extracted from such scans, then further analysis can be performed (calculating the volume of an organ for example). Therefore, the ability to segment medical scans into anatomical features can aid in diagnosis.
The purpose of our program SIMI is to aid clinicians or any other professionals whose line of work involves the use of medical images in performing segmentation on such type of images. Furthermore, it provides the user the ability to perform this process manually or by using semi-automated algorithms. These characteristics improve the precision of the resulting image.
Another use of SIMI is allowing the user to load a series of scans in order to observe the progression or the regression of a disease in organs. This is done by comparing the intensities of the infected organ(s) and their size(s) through the series.

We wanted to make SIMI as user-friendly as possible. The currently available medical imaging software packages are hard to use and require spending a great amount of time in learning how to use their features. SIMI presents the advantage of being easy to use by providing the user with an intuitive interface and requiring him/her to follow as few steps as possible.

1.1 Background

We incorporated some features that worked well in the existing segmentation tools: the orthogonal view (from ITK-SNAP) because it helps navigating through the image, Livewire(from Turtleseg). Also after using these tools we were able to make better decisions in designing SIMI. ITK-SNAP's 3D segmentation algorithm is confusing to use (it involves too many steps) and also it has a tendency to select regions outside the desired area of interest. We decided to use another algorithm which performs better(Connected Threshold). Also ITK-SNAP's interface is not very intuitive and requires too many steps in processing user input, which renders it difficult to use.

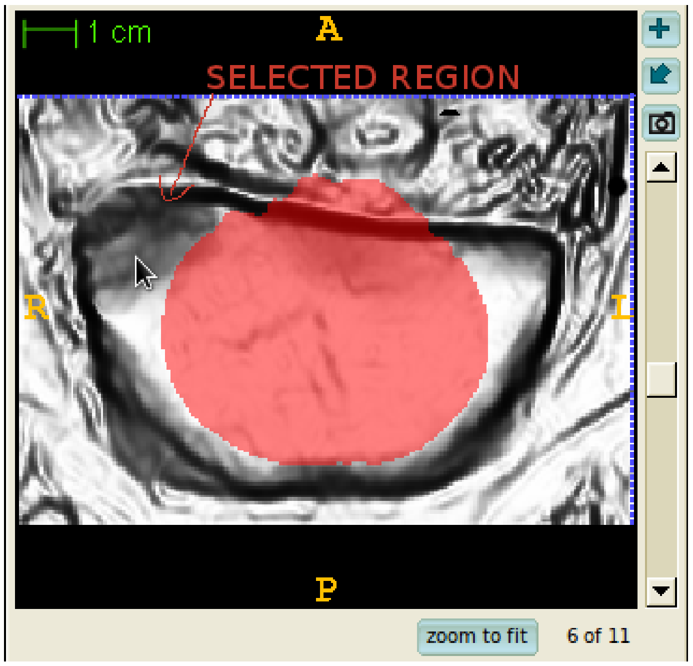


Figure 1.1: It can be seen that the segmentation mask has already "leaked out" of the desired region.

1.2 Choice of Technologies

1.2.1 Segmentation and Visualisation Libraries - ITK and VTK

As our project required providing the user with a means of segmenting medical images, it was important to find a library that provided a means of doing this, as trying to implement such algorithms ourselves would be difficult, as none of us had any experience with Computer Vision algorithms. It was essential to be able to display an image, and the results of the segmentation, and so use of a graphics library would be preferable to implementing our own from scratch, which would be beyond the scope expected for the project.

We were recommended by our supervisor to make use of the C++ ITK and VTK libraries, as it allowed us to make use of segmentation and visualisation algorithms we would need in order to meet the basic functionality requirements for our project. There was also some facility for the two libraries to interface with each other, via the project "ItkVtkGlue", which would be useful for converting the results of a segmentation from ITK to a form VTK could display. We therefore made use of these libraries.

1.2.2 Language Choice - C++

The ITK and VTK libraries provided various language bindings: VTK provided support for Java, Tcl/Tk and Python, while for ITK, the only supported language binding was Python. We chose not to use python, as we preferred to use a language we had some experience using, and learning a new language while developing the project would no doubt have slowed down progress in the initial stages.

We also found a patch for VTK and ITK, via ”VTK.NET” and ”ManagedITK”, which add bindings for .NET languages. This led us to consider using C#. However, as one of our aims was for the project to build and run in a Linux environment, we would have had to make use of the Mono framework. We would have also had to figure out how to compile VTK and ITK to support this, which would have added to the time for setting up a development environment before we could start work on the project.

We eventually decided on C++, as we felt it would be more reliable to use the native API to the libraries we were making use of. Finding support was also an issue, as when we encountered problems during making our project, we would have had to consider whether the problem was due to incorrect use of the libraries we used. We felt that increasing the number of libraries our project depended on would make it more difficult to debug.

1.2.3 GUI Library - Qt

For developing the GUI, we decided on Qt, as it had cross-platform support. We also briefly looked at using MFC. However, because one of our aims was for the project to work in Linux, we decided against this.

One of our goals was for the project to run in a Windows and Linux environment, and thus using a cross-platform GUI library was essential. Qt appeared to be one of the more mature and fully-featured cross-platform GUI libraries, and it also featured a tool - Qt Designer - which allowed for easy creation of the layout and look of the GUI elements in a window. Qt facilitates the production of easily maintainable and reusable code.

1.2.4 Version Control - git

The version control systems we considered using were: subversion, mercurial and git.

We opted to not use subversion, as it lacked the facility of local (off-line) commits. We felt this would be a useful feature to have, as it would enable us to make small regular commits locally to take advantage of the security a versioning system offers, in terms of being able to undo changes or inspect a previous version.

We chose git over mercurial, as some of our group members had used git in a previous project, and we felt that having some group members familiar with the tool would speed up the process of learning how to use it for group members that were not familiar with it.

Deciding on a place to host a central repository was considered. We compared two services that offered such hosting: bitbucket and github.

We chose bitbucket, as it offered a free private account with up to 5 users (the number of members in our group), and we were unsure at the beginning of the project whether there would be another group assigned to the same project as us, and we therefore thought that a private account was preferable.

1.2.5 Build System - CMake

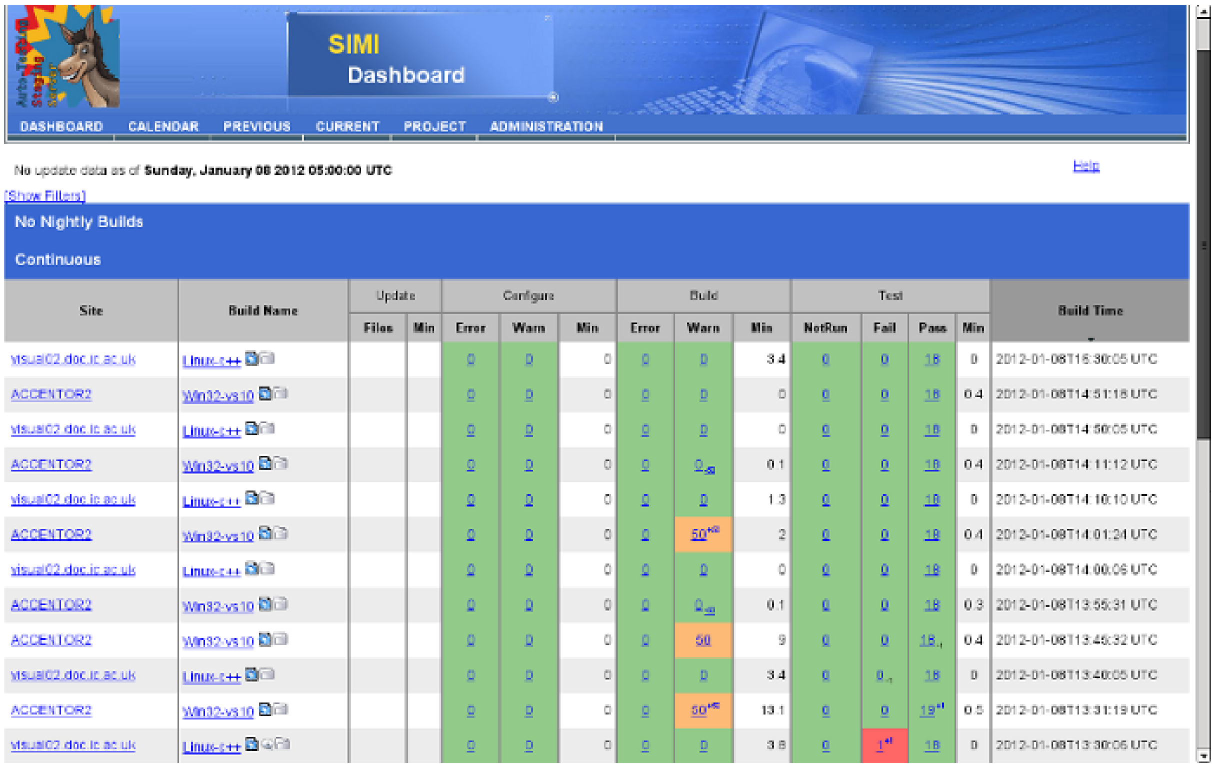
Our project required the use of a wide variety of libraries from different vendors. In order to simplify the management of linking these libraries, and to achieve our goal of cross-platform development, we enlisted the help of a build system. As using the VTK and ITK libraries required making use of the cross-platform CMake build system, we considered using this. After some research, we found that it generated Makefiles for a Unix environment, and *Visual Studio 2010* project files for use in a Windows environment.

CMake works by configuring the project within a file named `CMakeLists.txt`. Within this file, you specify the source files which make up your project, and list the libraries which the project depends on. The `CMakeLists.txt` file was relatively simple, and it simplified linking the ITK and VTK libraries to our project. Thus, we opted to use this to build our project as well. To build the project, one would run CMake against the `CMakeLists.txt` file. CMake would use a number of built in scripts to attempt to locate the libraries specified in the configuration file, and prompts the user if they cannot be found so that they may be set manually. Additionally, the target compiler is tested to make sure it is suitably able to compile the project. CMake will then generate a program maintenance file for the target platform. On Linux, CMake would generate a Makefile, and on Windows, CMake would generate a Visual Studio 2010 solution file. The program may then be built by a native compiler on the target platform.

1.2.6 Continuous Integration Server - CTest & CDash

One of the goals of our project was to make our program function correctly on Linux and Windows. This meant that we needed to test our tool on both platforms. We decided to set up a Continuous Integration Server, which would build and test the project on both platforms, and display the results on an overview webpage. The webpage would allow all group members to verify that the project was functioning correctly on Windows and Linux. We decided this was necessary, as each group member would be primarily working on one of the platforms (eg. Linux), and would rarely use the other.

CMake provides a unit testing system named CTest. In addition, it also provides a web based dashboard, named CDash, which displays the results of recent runs of CTest from various build servers. Each unit test within is specified in the `CMakeLists.txt` file, by giving a test executable and some arguments. The test is regarded as passed if the return value is 0.



No update data as of **Sunday, January 08 2012 05:00:00 UTC**

No Nightly Builds

Continuous

Site	Build Name	Update		Configure			Build			Test				Build Time
		Files	Min	Error	Warn	Min	Error	Warn	Min	NotRun	Fail	Pass	Min	
visual02.doc.ic.ac.uk	Linux-c++			0	0	0	0	0	3.4	0	0	18	0	2012-01-08T16:30:05 UTC
ACCENTOR2	Win32-vs10			0	0	0	0	0	0	0	0	18	0.4	2012-01-08T14:51:18 UTC
visual02.doc.ic.ac.uk	Linux-c++			0	0	0	0	0	0	0	0	18	0	2012-01-08T14:50:05 UTC
ACCENTOR2	Win32-vs10			0	0	0	0	0_{-50}	0.1	0	0	18	0.4	2012-01-08T14:11:12 UTC
visual02.doc.ic.ac.uk	Linux-c++			0	0	0	0	0	1.3	0	0	18	0	2012-01-08T14:10:10 UTC
ACCENTOR2	Win32-vs10			0	0	0	0	50^{+50}	2	0	0	18	0.4	2012-01-08T14:01:24 UTC
visual02.doc.ic.ac.uk	Linux-c++			0	0	0	0	0	0	0	0	18	0	2012-01-08T14:00:06 UTC
ACCENTOR2	Win32-vs10			0	0	0	0	0_{-50}	0.1	0	0	18	0.3	2012-01-08T13:55:31 UTC
ACCENTOR2	Win32-vs10			0	0	0	0	50	9	0	0	18_{-1}	0.4	2012-01-08T13:45:32 UTC
visual02.doc.ic.ac.uk	Linux-c++			0	0	0	0	0	3.4	0	0_{-1}	18	0	2012-01-08T13:40:05 UTC
ACCENTOR2	Win32-vs10			0	0	0	0	50^{+50}	13.1	0	0	19^{+1}	0.5	2012-01-08T13:31:19 UTC
visual02.doc.ic.ac.uk	Linux-c++			0	0	0	0	0	3.8	0	1^{+1}	18	0	2012-01-08T13:30:06 UTC

Figure 1.2: Screenshot of our continuous integration CDash displaying test results

The above screenshot shows our overview web page. As you can see, we used two build machines, `ACCENTOR2` (Windows 32bit), and `visual02` (Linux 64-bit). The right hand columns show the number of tests which passed or failed. The screenshot shows a situation (the bottom two rows) where a test failed on Linux, but passed on Windows, allowing us to quickly detect that there was a problem on Linux. Clicking the number of tests passed/failed displays detailed output from the unit test.

Our continuous integration server was operated by a series of scripts. BitBucket was configured to send a notification to the DoC web servers each time a commit occurred. This in turn sent a notification to our Linux and Windows build servers, which then checked out the latest git revision and built and tested the code. The results of these tests were then submitted to the DoC web server, which saved the results of the tests in a database, so that they may be inspected using the CDash interface.

1.2.7 Testing Framework - googletest and CTest

Unfortunately, CTest proved to be relatively basic, in that you could only specify an executable and some arguments to run, and CTest would inspect the return value. In order to write viable unit tests, it would either require having a separate executable for each test, or adding boilerplate code to our test executable's main method to switch between the tests. In addition, all the result of each assertion had to be tracked manually throughout the test, in order to ensure that the correct return value is given.

We decided to supplement our use of CTest with a unit testing framework. We considered two options: QTest (Built into Qt), and the Google C++ Testing Framework.

As our main goal was to reduce the amount of overheads involved in managing our unit tests, we chose the Google C++ Testing Framework. This was because it provided a macro for tighter integration with CTest, and it was possible to combine all the tests into one executable. The framework also provided a main method for the test executable, which handled the arguments and the invocation of each test. The CTest macro provided would automatically add each test to CTest, by providing the correct arguments to Google's main method. It was important that each unit test existed in CTest, as our CDash displayed the results of CTest, and not the Google C++ Testing Framework. Without this capability, all the Google Tests would have shown up as one unit test on the dashboard, providing a reduced level of feedback.

We rejected QTest, as it required each test case (group of tests) to run as a separate executable, and did not provide a macro, meaning you would need to add each test case to the `CMakeLists.txt` file manually. In addition, CTest/CDash would have only displayed the result of each test case, rather than each unit test.

1.2.8 IDE - Visual Studio 2010

The build system CMake we had chosen could generate Unix Makefiles for a Linux environment, and Visual Studio 2010 project files for a Windows environment. We therefore had to choose Visual Studio 2010 as an IDE to build and test the project in Windows.

1.3 Project Requirements and Extensions

1.3.1 Key Requirements

A pre-requisite to being able to segment medical images is being able to load them. There exist a number of different medical formats, and so it would be useful to be able to handle a few different image formats rather than just one. A facility to save the resulting segmentation was also essential.

- Load medical images of type DICOM and VTK. After performing a segmentation, the result will be converted into a VTK format.

Although we aimed to make use of semi-automated ITK segmentation algorithms, having manual control of the segmentation to make tweaks and changes would be expected by a user.

- Segment images using a manual pen/eraser tool, by colouring in 2D slices.

We opted for having 2D segmentation as a key requirement, as we assumed it would run faster than a full 3D segmentation algorithm. This would be important for user interactivity.

- Allow the user to perform 2D region growing on a 2D slice.

A useful reference for the user while segmenting an image would be how large the current segmentation is. The intention was that this would give the user an idea of whether the segmentation included what they expected, provided they already had an idea of how large the segmentation ought to be.

- Calculate the volume of a 3D segmented object by counting the number of voxels [1] it contains.

It was considered useful to provide a safety net in case the user forgot to save the result of their work.

- An auto-save feature. Periodically save any segmentations created.

1.3.2 Extensions

One use-case a user might expect would be being able to segment multiple features within an image. Thus we considered supporting this functionality.

- Allow user to perform multiple segmentations in one file, and set a colour and a label for each segmentation.

While the tools we chose to implement in the Key requirements would allow the user to produce a 3D segmentation, it would have to be done slice by slice (using the manual pen tool or the 2D segmentation tool). Making use of a 3D segmentation algorithm would further automate the segmentation process for the user, and thus save their time.

- 3D region growing.

There are several different algorithms that attempt to segment features of interest in an image. These all work with varying degrees of success, depending on the type of image, image quality, resolution, sensitivity to any user information given, etc. Thus it would be useful for the user to have a fall-back to use a different algorithm, should one produce spurious results for a given image.

- Implement an alternative algorithm to perform segmentation (using one of the following algorithms: Geodesic Contours, Watershed, Level Set, Livewire)

The most useful safety net to provide a user would be an undo facility, so they could recover from any mistakes they made to the segmentation while using the program.

- Have a (bounded) undo facility that can undo the last n user actions.

With the aim of extending the project to be used by a wide audience, including Linux as a deployment environment would allow for more users to use our program.

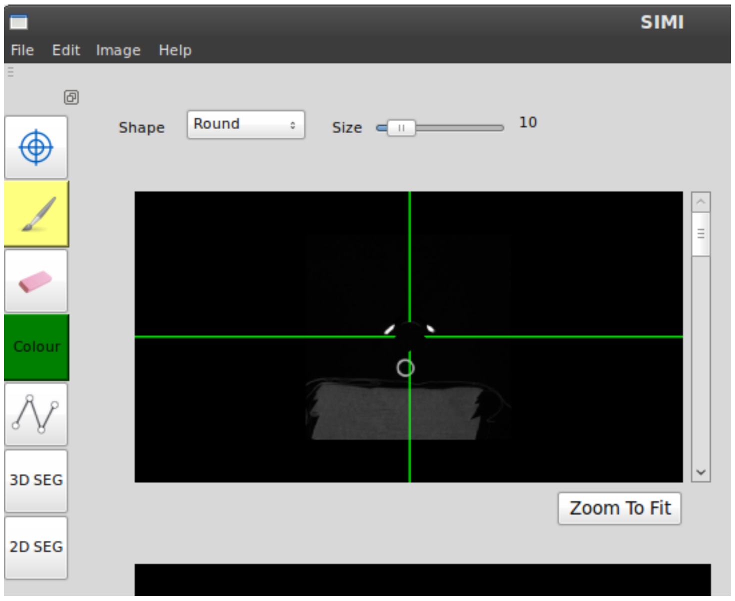
- Build project on Linux.

1.4 Design

1.4.1 GUI

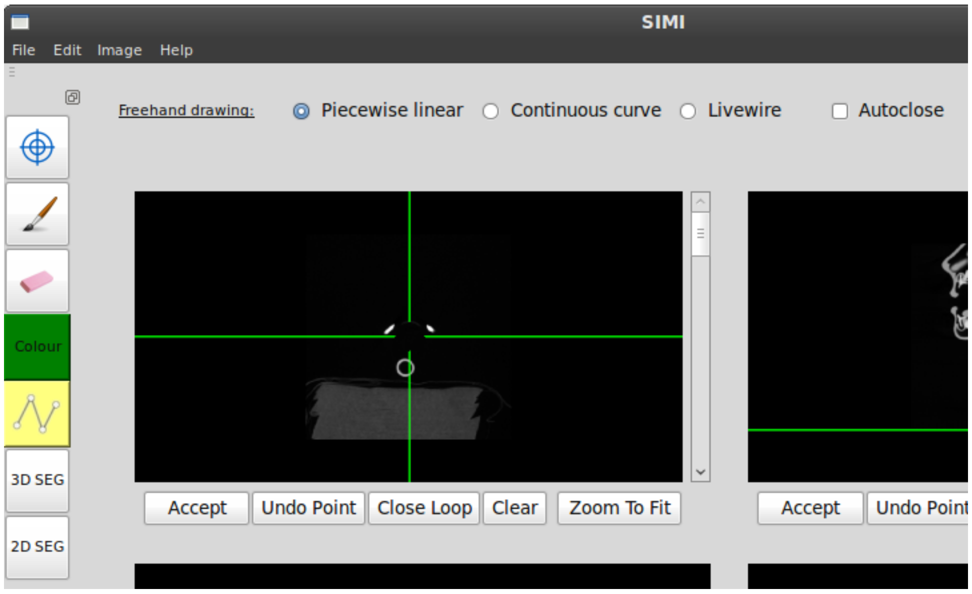
One of our intentions was to make the process of using SIMI as easy as possible. Thus from the GUI point of view, by displaying a QToolbar on the left hand side with menus for each tool appearing only after the corresponding button had been pushed, makes it easier for the user to perform the action that he/she chooses rather than being distracted by the display (had all the options been statically presented on the screen). Also, all the buttons display intuitive icon images and explanations of their purpose.

[1]Voxels, or Volumetric Pixels, are analogous to pixels, but in 3D.



Shape: Round. Size: 10

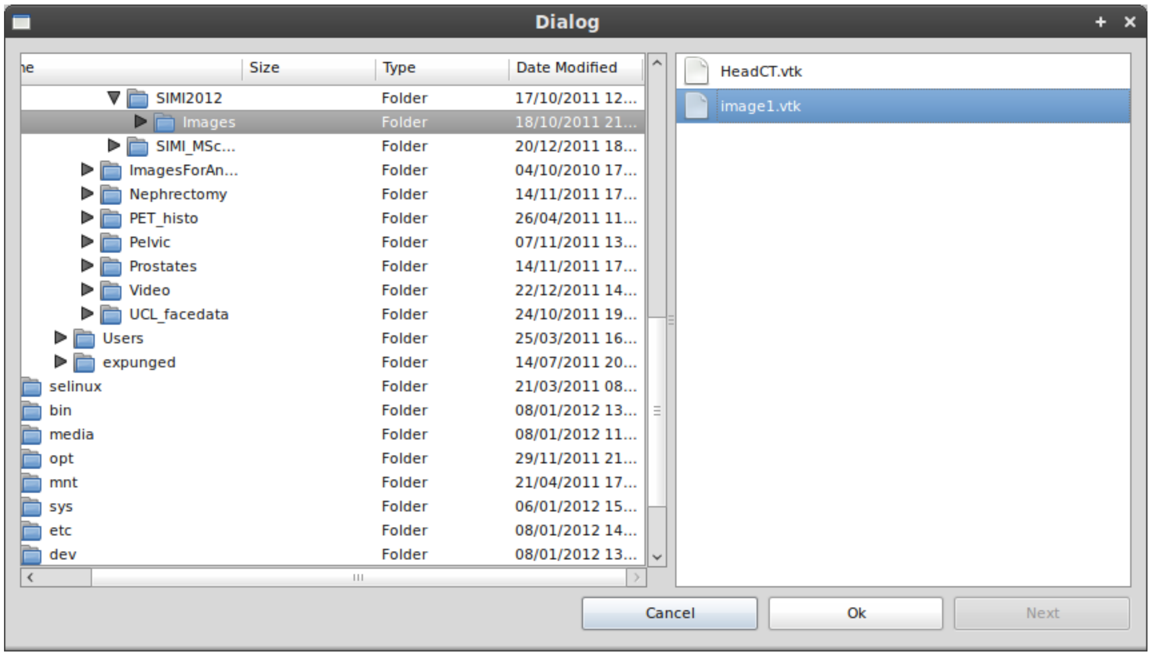
Figure 1.3: The paint tool is selected and the corresponding options appear on top of the orthogonal view



Freehand drawing: [✓ Piecewise linear] Continuous curve Livewire ☐ Autoclose

Figure 1.4: The selected tool is now the Line tool, and as you can see different options are displayed

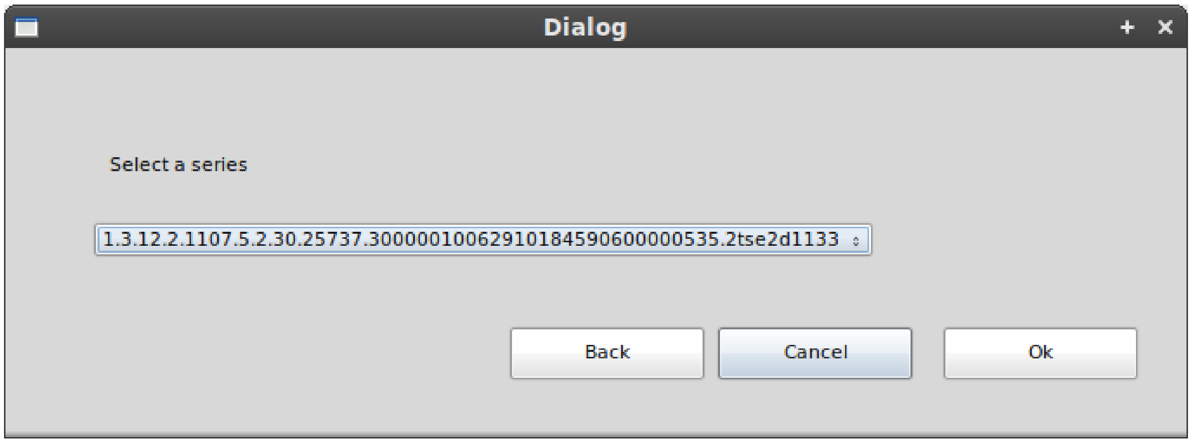
Another GUI design decision that was made to aid usability was creating the Open dialog to currently accept .vtk and .nii images (and just use one dialog) or DICOM (for these images another dialog will be opened to select a series). We felt that not forcing the user to use multiple dialogs for opening the first two mentioned types of images would make our program more user-friendly. This would have been an issue with using the native QFileDialog class.



ne	Size	Type	Date Modified
SIMI2012		Folder	17/10/2011 12...
Images		Folder	18/10/2011 21...
SIMI_MSc...		Folder	20/12/2011 18...
ImagesForAn...		Folder	04/10/2010 17...
Nephrectomy		Folder	14/11/2011 17...
PET_histo		Folder	26/04/2011 11...
Pelvic		Folder	07/11/2011 13...
Prostates		Folder	14/11/2011 17...
Video		Folder	22/12/2011 14...
UCL_facedata		Folder	24/10/2011 19...
Users		Folder	25/03/2011 16...
expunged		Folder	14/07/2011 20...
selinux		Folder	21/03/2011 08...
bin		Folder	08/01/2012 13...
media		Folder	08/01/2012 11...
opt		Folder	29/11/2011 21...
mnt		Folder	21/04/2011 17...
sys		Folder	06/01/2012 15...
etc		Folder	08/01/2012 14...
dev		Folder	08/01/2012 13...

HeadCT.vtk
image1.vtk

Figure 1.5: Dialog for opening .vtk, .nii files



Select a series

1.3.12.2.1107.5.2.30.25737.30000010062910184590600000535.2tse2d1133

Figure 1.6: Dialog for opening DICOM files

1.4.2 Segmentation

Segmentation is a major part of the program. The user can perform a number of different segmentations, and all change the segmentation mask. Each type of segmentation has its own class and methods corresponding to the interactions required for the segmentation. Figure 1.7 shows the inheritance diagram for the Segmentation class. For example, the DrawingSegmentation class provides methods for drawing points, lines and paths on the segmentation mask. For each segmentation class there exists a QWidget which interacts with the methods of the class. For example the SegThresholdsMenu has two QSlider objects which call the setThresholds method of a ConnectedThresholdThreeD class.

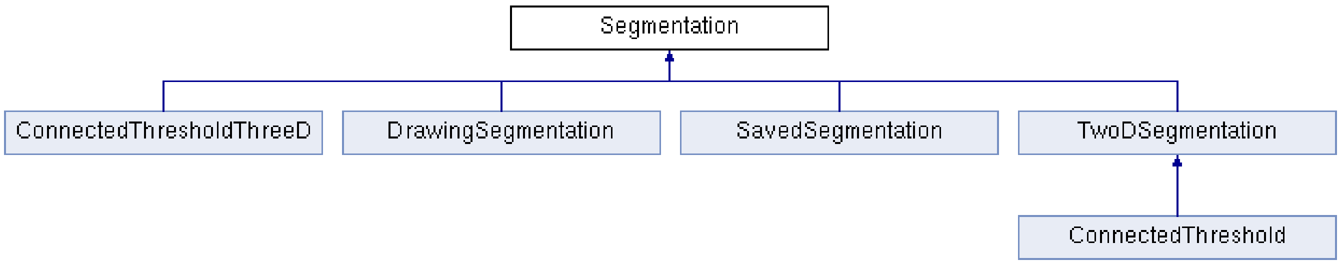


Figure 1.7: Inheritance diagram for Segmentation

1.4.3 Image Data

A major component of the system is the SimiImage class: it contains all the image data, mask data and methods for querying information about the image. As shown in the inheritance diagram (figure 1.8) we had a couple of subclasses for the SimiImage. The SimiImage class is constructed when loading a VTK or similar image, whereas the SimiDicomImage is used when loading a series of images in the Dicom image format. During development it was found that these separate constructors were creating duplication of code and so a SimiImageFactory class was created which centralised the process of checking the type of the image and creating the corresponding object.

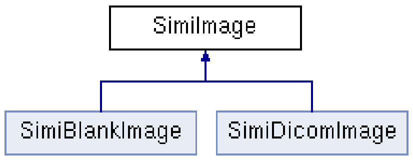


Figure 1.8: Inheritance diagram for SimiImage

1.4.4 Orthogonal View

The orthogonal view is a QWidget which contains three ViewSeg classes. The ViewSeg class is itself a QWidget which has methods for manipulating what is displayed. The ViewSeg class has methods for manipulating the camera location, manipulating the location of the cursor and setting the active tool. A tool dictates the behaviour of the mouse (e.g. the DrawingTool makes the mouse act as a paintbrush). We designed the OrthogonalView class such that methods called on an OrthogonalView would propagate to the ViewSeg objects inside. The Render method of the OrthogonalView calls the Render method on the three ViewSeg objects.

1.4.5 Tools

Each tool has its own interactions and methods pertaining to its purpose. Each subclass of SimiTool as shown in figure 1.9 has an execute method which gets called when the user performs a mouse event on one of the ViewSeg objects. The tool will be told which ViewSeg sent the message, the mouse event ID and any data associated such as mouse coordinates.

Only one tool can be enabled at one time, and when the user selects a different tool from the menu then setTool method will be called on the OrthogonalView object which in turn calls setTool on each ViewSeg object inside of it.

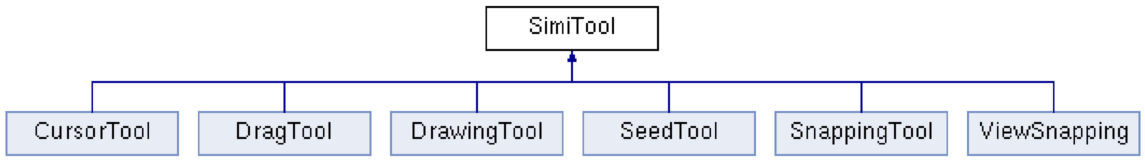


Figure 1.9: Inheritance diagram for SimiTool

When we began designing SIMI, one of the first decisions we had to make was to choose which mask annotating tools we were going to implement.

Drawing Tool (Paintbrush and Eraser)

The first and obvious tool we decided to implement was the Drawing Tool. The user had to be able to manually mark the mask at any point he wanted. Respectively, he would have to be able to "unmark" the mask as he sees fit. That is why we decided to add a two modes to the Drawing Tool. One to allow the user to add to the mask, and another to delete sections. These are available via the GUI as the Paintbrush and the Eraser respectively.

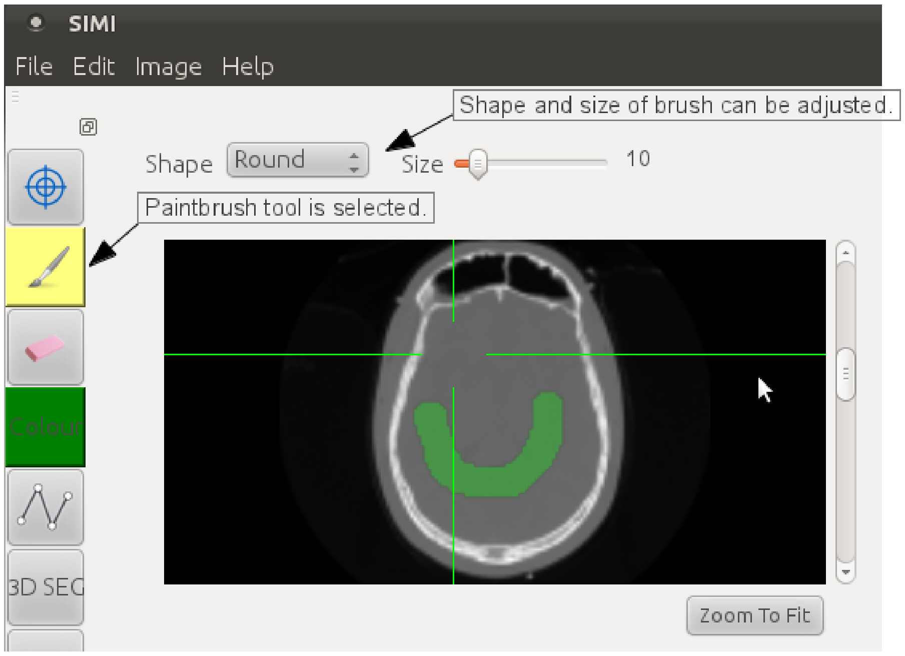

SIMI
File Edit Image Help
Shape and size of brush can be adjusted.
Shape Round
Size 10
Paintbrush tool is selected.
Colour
3D SEG
Zoom To Fit

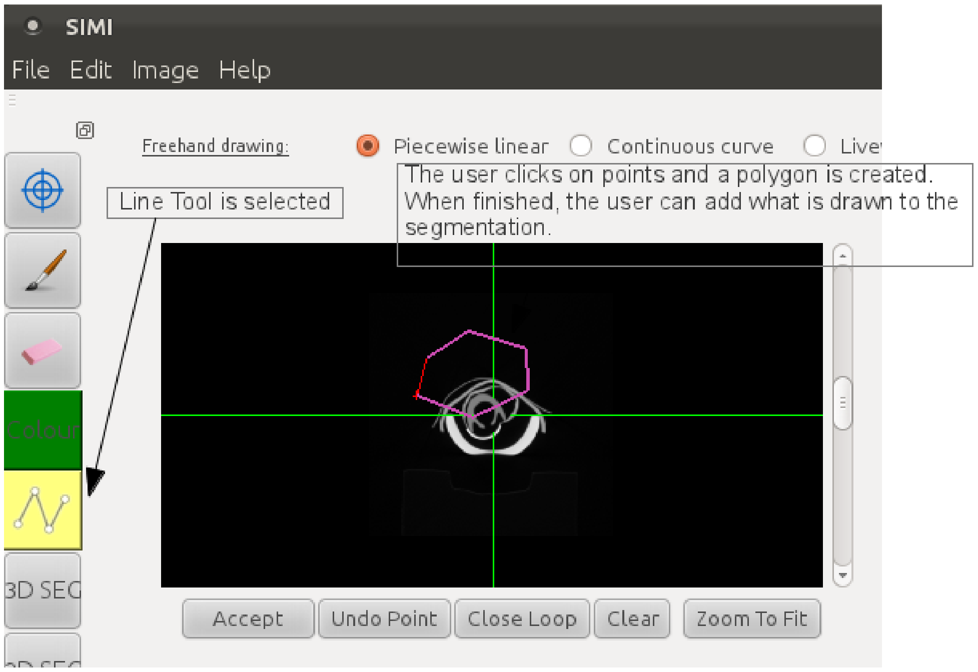
Figure 1.10: Use of the Drawing Tool.

Line Tool

The main drawback of the manual annotation using the Drawing Tool is the low accuracy. The low accuracy is because of two reasons. First, and most important, is the low accuracy of the human hand. Additionally, marking a point using the Drawing Tool, also "paints" a small radius of points around it. That is why we decided to build the Line Tool. The Line Tool, is a tool that allows you to mark a large surface in an easy and more accurate way. The main functionality of the tool is that the user marks two points, and the tool connects the with a line. When a closed loop of lines is formed, then the tool can mark the whole surface on the mask.

The first mode we built for the Line Tool, the Linear mode, would connect the two points with a straight line. That is the simplest form of the Line Tool. Although one might rarely need to annotate a surface of straight edges, the main advantage is the very fast marking of big surfaces with a few clicks. Moreover, if one zooms in the image and places a big number of short straight lines, he can give the illusion of a straight curved line and so mark a surface, quite accurately.

This capability of the Linear mode of the Line mode, led us to formalise this feature and build a separate mode for it, calling it the Curve mode. The Curve mode has all the functionalities of the Linear mode. The difference is that this mode allow the user to click and drag so that he can connect two points with a curve. In reality the tool places a big number of very small straight lines, and so produces a curve.

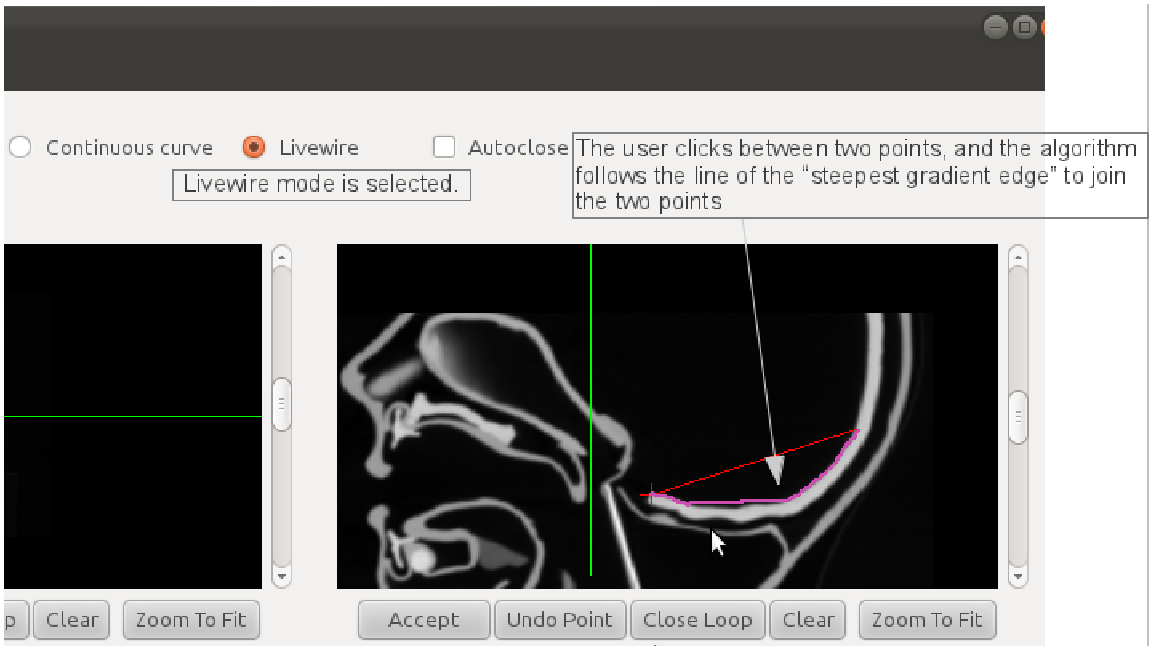


The user clicks on points and a polygon is created. When finished, the user can add what is drawn to the segmentation.

Figure 1.11: Making use of the Piecewise Linear mode of the Line Tool.

LiveWire mode for Line Tool

The next step, when deciding which tools to use, was to try to increase even more the accuracy. That is the reason we decided to introduce Livewire functionality to the Line Tool. The Livewire, using an algorithm (described in another section), helps in the selection of a surface of interest in a quick and accurate way, using mouse clicks only. That is the main reason why we decided to implement the Livewire.



Livewire mode is selected.

The user clicks between two points, and the algorithm follows the line of the "steepest gradient edge" to join the two points

Figure 1.12: Use of the LiveWire mode of the Line Tool.

Chapter 2

Software Engineering Issues

2.1 Algorithms Used

2.1.1 Connected Threshold

For our project we wanted a semi-automatic method for segmenting images. There are a number that have been developed but we wanted to use one which had been developed and tested as part of the Insight Toolkit. We found that the Connected Threshold algorithm had been used successfully by a number of other people who were looking at segmenting medical images[1].

The 3D and 2D Connected Threshold segmentation algorithms used in our project make use of region growing with one seed point and a maximum threshold and minimum threshold. Region growing is an iterative process which looks at adjacent pixels to determine if they are connected. In the Connected Threshold algorithm, a pixel is connected to another if its intensity falls within the range of the configured maximum to minimum threshold values. The algorithm starts with the region being that of the seed point and iteratively increases this region by adding the adjacent connected pixels.

The 2D Connected Threshold algorithm was relatively fast, this allowed us to make it interactive with the GUI. Whenever the slider values for minimum and maximum threshold values are changed the algorithm is re-run and updates the GUI. The 3D Connected Threshold algorithm was much slower, we only ran the algorithm when the user clicked the "Compute" button.

On the sample images we used the algorithm performed very well, and if you knew the specific thresholds you could view separate skin and bone structures. 2.1, 2.2

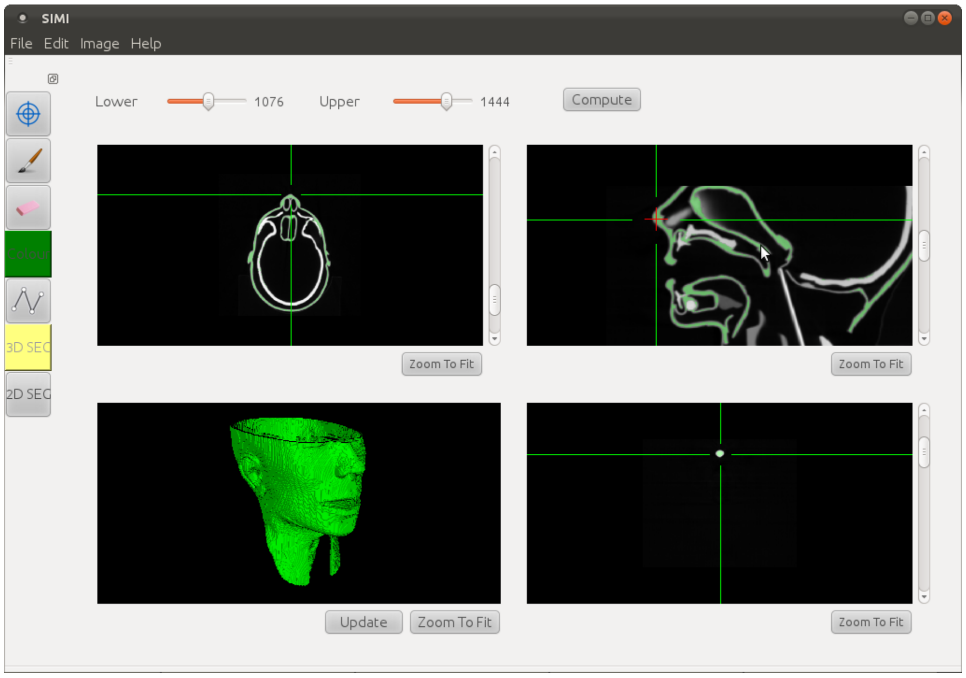


Figure 2.1: Applying "Connected Thresholds" using an intensity range of 1000-1500

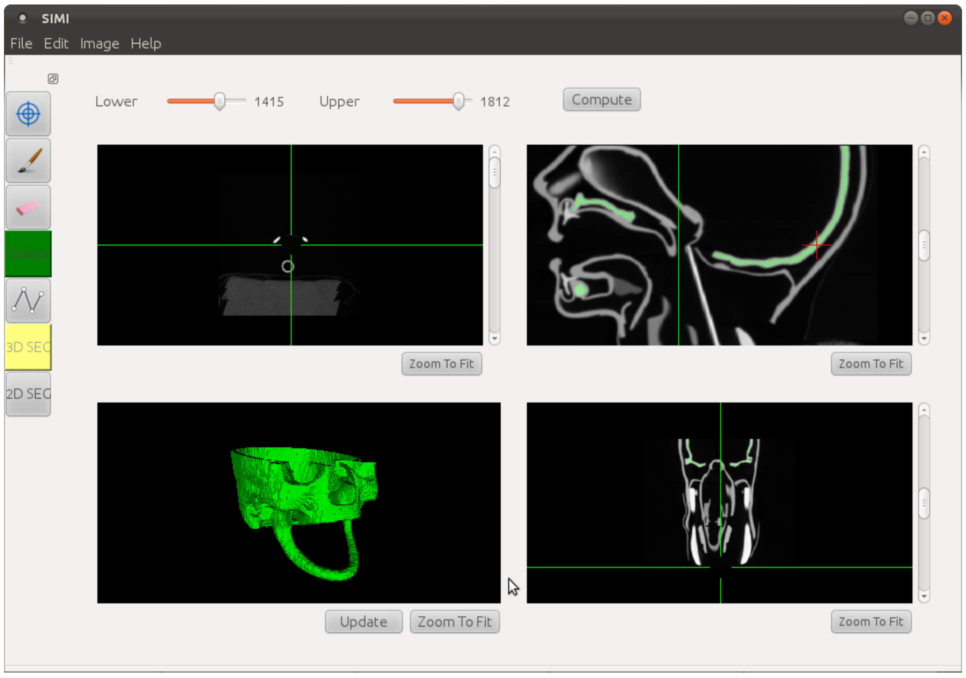


Figure 2.2: Applying "Connected Thresholds" using an intensity range of 1400-1800

2.1.2 LiveWire

An important part of SIMI is to give the user the chance to manually segment an image. One of the methods that have been implemented is the "Livewire mode" of the Line Tool. The Livewire algorithm, as used in our program, will connect two points on a 2D image by following edges of great colour differences. It is also known as Intelligent Scissors.

In reality, the Livewire looks for the minimum cost path on a directed weighted graph. To do that, the Livewire makes use of two main sub-algorithms: One to produce the graph, and one to find the minimum cost path. To find the minimum path, an implementation of Dijkstra's algorithm is used.

To produce the graph, each pixel represents a node on the graph. Directed edges from each pixel connect them to all their neighbour pixels. The weight of each edge is calculated by a cost function that depends on three variables: the gradient of the image, the zero crossing, and the boundary smoothness constraint associated with the gradient direction at each pixel.

When the graph is produced, Dijkstra's algorithm finds the path wanted.

2.1.3 Winding number algorithm

The Line Tool works by defining a set of adjacent lines that form a closed loop. Then the user should be able to copy the whole area on the mask. To decide whether a point is in a closed curve or not, a "Point in closed curve" algorithm is needed. The one we used was the Winding number Algorithm.

The algorithm works by computing the winding number of a point, with respect to the closed curve. If the winding point is non-zero, then the point lies within the curve. The winding number of a point in respect to a closed loop is an integer representing the number of counter-clockwise circles that are travelled around the point by the curve.

Although the winding number algorithm is a bit costlier than the Ray casting algorithm that is also widely used, it has the advantage that it allows for crossing lines on the closed curve without ruining the result.

2.1.4 Marching Cubes

The goal of our program is to help the user produce a segmentation from a medical image. To verify the segmentation produced, it is useful to be able to visualise it in 3D.

The segmentation produced is a 3D mask image comprised of a number of voxels. Voxels are the 3D analog of 2D pixels. While a 2D pixel is considered a fixed size square, a voxel is considered as a cuboid. Several voxels of different colours/intensities can be used to build up an image.

To convert this voxel image into a polygonal mesh that VTK can render, we make use of an algorithm known as "Marching Cubes". The purpose of the "Marching Cubes" algorithm is to produce a polygonal mesh from an image consisting of voxels. A voxel is processed by the algorithm if its intensity is greater than some predefined limit. This helps to select a skin to generate among an image of different intensity voxels.

The algorithm takes each voxel in turn and considers its neighbours (other voxels that touch it at one of its 8 vertices). Depending on the configuration of its neighbours, a piece of a polygonal mesh, with a certain configuration, is used to represent that voxel. The process is repeated, until all voxels have been examined.

The result is a topologically consistent polygonal mesh (that is, there are no tears in the mesh produced). This may then be used by VTK's rendering pipeline to be shown to the user.

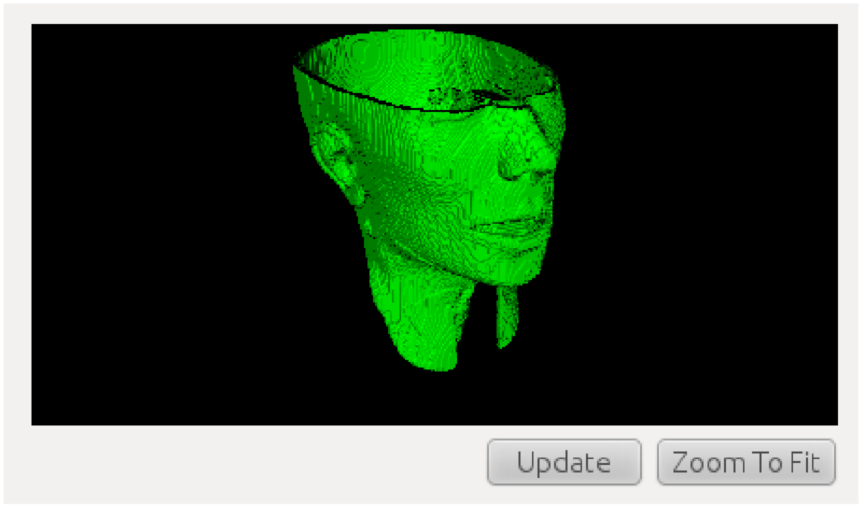


Figure 2.3: Result of applying "Marching Cubes" algorithm on an image of voxels.

2.2 Technical Challenges

2.2.1 Memory Leaks

The size of medical images is usually large, as the images are typically in three dimensions. To process images, it is necessary to load them into memory. As memory is finite, it was important that our program managed memory efficiently. In C++, there is no automatic garbage collection, which makes the process of managing memory more difficult, as each object must be told specifically to delete itself. In addition a fatal crash occurs, if an object is deleted twice, further increasing complexity.

A solution to this problem is to make use of smart-pointers. ITK and VTK[4] both provide implementations, and both implementations are used within SIMI. Both implementations work by counting the number of times they have been referenced. When the smart-pointer is dereferenced, and the number of references reaches 0, the object deleted, and the memory it occupied is reclaimed. This solves the problem of double-frees, and memory leakage, as the process of deleting objects becomes semi-automated.

Whilst smart-pointers provide an effective way to protect against memory leakage, it would be inefficient to use smart-pointers on all objects. This is because a smart-pointer creates an overhead by effectively acting as a middleman to an object, and for smaller objects, which do not occupy much memory, this overhead may outweigh the benefit. This is especially true for small objects, which exist only for the scope of a single function, as the object can easily be managed within a function, and its lifespan may be very short. This means that some objects may need to be created and deleted manually in order to retain performance.

VTK provides a useful feature for finding leakages for these smaller objects. By enabling the VTK_DEBUG_LEAKS flag within VTK, the program will output a count of the number of instances of a class which are still in memory after a program has destructed. An example output from VTK_DEBUG_LEAKS is shown below:

```
Class "vtkXOpenGLRenderWindow" has 3 instances still around.
Class "vtkOpenGLScalarsToColorsPainter" has 15 instances still around.
Class "vtkImageImport" has 2 instances still around.
```

This has helped us to trace the source of some memory leaks, as we can find where these classes were initialised, and make sure we delete the objects appropriately.

2.2.2 Build System

The unit testing framework required there to be a separate test executable in order to run the unit tests. This provided a problem, in that the code available in the main program would also need to be accessible from the test executable. The usual method of adding source files to each executable in CMake, resulted in the compiler processing some portions of the source code more than once. This nearly doubled the build time of our project. In order to resolve this problem, we moved all the code from the main executable except from the main method into a static shared object. Each executable (the main executable, and the test executable) then linked its main method onto the shared object. This meant that the code would only need to be compiled once, but would still be available in both executables.

2.2.3 Library Issues

Whilst configuring our project to link with the numerous libraries (ITK, VTK, Qt, etc..) required for our project, we encountered a number of issues with operating system supplied binaries. For example, Ubuntu's version of VTK was not built with Qt support, and was therefore unable to display graphics within a Qt interface. To solve these issues, it was decided that we would build and maintain our own versions of the libraries, so that they are configured correctly for our project. This solves the immediate problem of missing features from the binary versions, but may affect the ability for the program to be deployed elsewhere. It is therefore likely that some of our dependencies would need to be distributed along with our project, which could possibly cause licensing issues. Fortunately, ITK and VTK are both licensed under the BSD License, and Qt is licensed under LGPL (Lesser General Public License). Both these licences allow the redistribution of unmodified versions of the libraries concerned.

2.2.4 Orientation issue between ITK and VTK libraries

Another library issue we faced involved viewing the image. We made a design decision to make use of ITK's image object, as opposed to VTK's image object to store our image and mask. This was because we would have to pass an ITK object to and from the ITK library when we wanted to perform a segmentation. Thus it made sense to store our image internally as the same format.

However, to display the image, we had to convert the itk image to a vtkImageData object, via an itk filter. There was a problem with the image displayed, as pointed out by our supervisor, in that the orientation of one of the views was the opposite to what you would expect. The image appeared flipped in the XZ plane.

After searching the itk and vtk message boards, it appeared that this problem was frequently encountered. One of the proposed solutions was to adjust the orientation of the camera to correct this[2]. All that would be required would be to change the camera position, such that the view was flipped in the XZ axis to show the correct orientation.

2.2.5 Positioning of the Cursor

One of the main problems we faced fairly early on the project were the different coordinate systems supported by VTK. All the different structures in VTK were built using different coordinate systems, with extremely limited documentation. The creators themselves have admitted a lack of support on this issue.

After researching the problem, we decided to use a coordinate system as our default system and transform all others to that. This did not prove to be an easy task, as no transformation functions, or coordinate system definitions were provided. We did manage, though, to build them ourselves.

2.3 Risks

2.3.1 Libraries

A problem we encountered was making use of the large VTK and ITK libraries. When trying to implement a feature, it was not always obvious which classes to study and make use of. The documentation available provided details on each class and its usage, but it proved difficult to find an overview of the structure of the libraries, which made making design decisions regarding how to interface with the libraries difficult at times.

We found the advice of our supervisor helpful on such occasions, as he was familiar with the libraries. Also searching VTK and ITK's mailing lists for questions regarding implementing certain functionality was useful.

2.3.2 2D segmentation

We were warned by our supervisor that the process of extracting a 2D slice from a 3D image and then inserting the same slice back into the image might be a difficult process. This was an important part of the software as it was needed as part of the 2D connected threshold algorithm.

After trying a number of functions available as part of the Insight Toolkit we found that extracting a 2D slice was an easy process. Trying to insert a 2D slice was a harder process however and we found that it was easier to write some code from scratch to set the voxels in the 3D image.

Adrian spent a number of weeks looking at the problem while doing other tasks because of the anticipated difficulty. We felt that by trying to tackle the problem early on in the project we allowed plenty of time for finding the solution and the risk of not completing this crucial component was reduced.

2.4 Collaboration Issues

During the second week of the project, one of the members fell ill for a few days, and as a result had to miss two meetings in a row. This caused him difficulties compiling the libraries that we were going to use. We managed to solve this by helping him compile all the libraries and setting up Visual Studio quickly, so he could catch up.

Towards the end of term it was on occasion difficult to find times when we were all free to meet to discuss the project. Typically we aimed to meet on a Monday, in order to prepare a version with added features to show at our next supervisor meeting on a Wednesday. Because we could not typically find times when we were all free on Monday or Tuesday, our compromise was to meet after 5pm on Mondays to prepare for the next meeting with our supervisor.

2.5 Testing Methods and Validation

2.5.1 Unit Testing

As mentioned in the Choice of Technology section, we made use of unit testing in order to verify that certain portions of the system worked as expected. This also gave the added benefit of testing the code on different operating systems. We planned to use unit tests wherever possible.

Unit tests were considered unnecessary for some portions of the code, which simply construct objects for use with algorithms within ITK/VTK. The results of operations such as these (Marching Cubes, Segmentation etc) were often inspected visually to ensure correct behaviour. However, at certain places we performed some operations within SIMI, and some of these were covered by unit tests.

For example, while starting work on the Undo feature, writing unit tests in conjunction with writing the code made for a useful safety barrier against regressions when developing the feature. However, due to time constraints, the feature wasn't fully implemented. Nevertheless, the use of a unit test for auxiliary functions that would have been used by the undo feature managed to find a bug in the implementation of a function that compared whether two images were equal.

2.5.2 Manual Verification

One of the difficulties encountered during our project was mapping the mouse cursor position on each orthogonal view to the correct voxel within the image. This was not trivial, as each orthogonal view allowed the user to zoom and pan, and in addition, the size of each voxel could vary.

In order to verify that our solution performed as expected, we developed a tool to generate test images with voxels varying in size in each dimension. A resulting image generated by the tool would have the appearance of an equal sized cube, made up of a different number of voxels on each side. Each side of the cube would be made equal length by changing the voxel size for that dimension. The image would also have the property that no two adjacent voxels would have the same colour.

This allowed us to check that the cursor mapped to the correct position by clicking within a square, and checking that the other orthogonal views updated to the correct voxel. In addition, we wrote the co-ordinates of the selected voxel to the terminal, and verified that the co-ordinates corresponded to the selected voxel.

Another test performed using this image was boundary checking. It was possible to click and drag within each voxel, and check that the other views only changed when the cursor left one coloured voxel, and entered another.

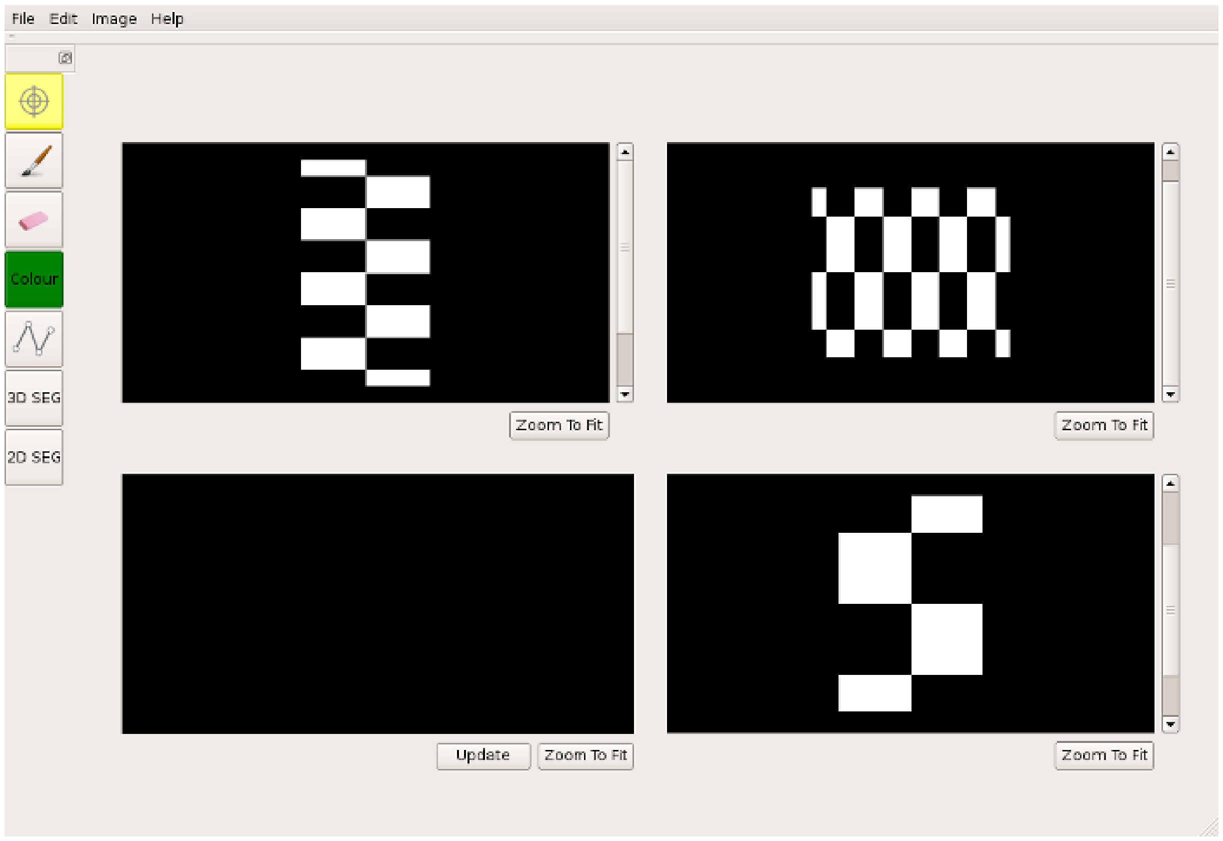


Figure 2.4: Screenshot displaying one of the generated checkerboard images used for verifying that mouse cursor position mapped to the correct voxel within the image.

In order to verify that we were displaying the orthogonal views correctly, we compared our views with the orthogonal views within ITK-SNAP. Using this method, on one occasion we found that one of our views was displaying upside-down. We were able to correct this problem and match ITK-SNAP's views, which allowed us to conclude that our views were working correctly.

2.6 Achievements and Member Contributions

2.6.1 Code Length

The table shown below contains the number of lines for .cpp files and does not include the lengths of the .h(header) files or .ui(used for the user interface). By using Qt Designer, automatic XML code was generated for the .ui files.

Class Name	Lines of Code
ConnectedThreshold	64
ConnectedThresholdThreeD	65

CursorTool	172
DragTool	30
DrawingSegmentation	227
DrawingTool	85
ImageLine	72
LiveWireImageFunction	266
LiveWireTool	72
SavedSegmentation	22
SeedTool	104
Segmentation	6
SimiBlankImage	17
SimiDicomImage	24
SimiImage	196
SimiImageFactory	33
SimiUtils	57
SnappingTool	59
TracingTool	86
TwoDSegmentation	128
View3D	130
ViewCursor	205
ViewSnapping	424
connected2dmenu	67
enclopdialog	118
erasermenu	53
imagedetails	106
main	22
mainwindow	522
opendialog	135
orthogonalview	114
paintbrushmenu	41
polygontoolmenu	82
segthresholdsmenu	78
setimageopt	56
settings	46
viewseg	336

2.6.2 Summary of Member Contributions

Below is a list of the features/functionality each group member contributed to the project:

- Adrian
  - Researching Connected Threshold

  - Taking 2D slices from a 3D image
  - 2D segmenting
  - 3D segmenting
  - Putting 2D slice back into 3D image
- Daniel
  - Autosave & Crash Recovery Feature
  - Load Recent Images Feature
  - Test Image Generator (Checkerboard Test Image)
  - Set up Testing Framework (Integration with CMake and CDash)
  - Continuous Integration Servers and Related Scripts (CRON / BATCH)
- Raluca
  - Open Dialog for loading images.
  - Setting the brightness of an image.
  - Display of image information.
  - MainWindow main menus, toolbar and menus for tool buttons.
- Savvas
  - Researching coordinate systems in VTK
  - Building the cursor and its correct display on the GUI
  - Completing the Drawing and the Eraser Tool
  - Line Tool
  - Implementation of the Livewire algorithm
- Shaan
  - Loading Images (DICOM and other formats: vtk, nii, etc)
  - 3D View of segmentation mask.
  - Image navigation in the Orthogonal View.

2.6.3 Log Book

Adrian	
Week 1 (17/10/11 - 23/10/11)	
1 hour	Setting up Git and discussing roles
Week 2 (24/10/11 - 30/10/11)	
1 hour	Setting up Wiki and first report
2 hours	Working on report and setting up CMake/CTest
Week 3 (31/10/11 - 6/11/11)	
2 hours	Working with Shaan with ITK/VTK/QT integration
2 hours	Researching and implementing 2D region growing
4 hours	Working with C++, inheritance and creating classes for segmentation
Week 4 (7/11/11 - 13/11/11)	
3 hours	Working with Raluca on viewseg class and series dialog class
2 hours	Cleaning up, refactoring code
3 hours	Getting orthogonal view working and transparent overlay mask
1 hour	Working on fixing code with events
2 hours	Trying to get VTK interactor working
Week 5 (14/11/11 - 20/11/11)	
4 hours	Working with vtk interactor and image intensities
4 hours	Working on writing segmentation to overlay mask
2 hours	Integrating the segmentation into the GUI
Week 6 (21/11/11 - 27/11/11)	
3 hours	Segmentation from 2D to 3D
1 hour	Working with Raluca integrating her changes
3 hours	Improving the segmentation
Week 7 (28/11/11 - 4/12/11)	
1 hour	Fix a number of bugs with segmentation
1 hour	Allow segmentation to be loaded from file
3 hours	Worked on paintbrush and changing the colour of the mask
3 hours	Worked on eraser
Week 8 (5/12/011 - 11/12/11)	
1 hour	Refactoring some code
2 hours	Investigating and fixing bug with drawing tool
2 hours	Writing some unit tests

Raluca	
Week 1 (17/10/11 - 23/10/11)	
3 hours	Researching best GUI practices
2 hours	Created the initial GUI design
2 hours	Setting up QT
Week 2 (24/10/11 - 30/10/11)	
1 hour	First report
3 hours	Familiarization with QT Creator (Tutorials on how to use it)
3 hours	Familiarization with QT Designer and building mock projects
Week 3 (31/10/11 - 6/11/11)	
6 hours	MainWindow (creation, enabled the Main Menu's actions, created shortcuts, created the Toolbar)
3 hours	Setup Build Environment (installed ITK, VTK, CMake)
2 hours	Merging the GUI with the rest of the project
2 hours	Created a widget for displaying the images.
Week 4 (7/11/11 - 13/11/11)	
7 hours	Second report
Week 5 (14/11/11 - 20/11/11)	
4 hours	Read *C++ GUI Programming with Qt4* by Jasmin Blanchette and Mark Summerfield
2 hours	Researching on how to determine the brightness in an image, classes that must be used
Week 6 (21/11/11 - 27/11/11)	
2 hours	Image Details (created the display)
3 hours	SimiImage (added intensity and rescale filters, created methods for brightness)
2 hours	Brightness Dialog (added sliders for brightness)
2 hours	MainWindow (methods for opening a dialog, added methods for displaying the Brightness Dialog)
7 hours	Open Dialogs (research on which classes would be most suitable, created one dialog for opening vtk, nii images and another for DICOM)
Week 7 (28/11/11 - 4/12/11)	
1 hour	MainWindow (added enabling, disabling functionality to actions and buttons depending if an image has been opened or not)
2 hours	Custom layout for mainwindow
4 hours	Image Details (added display functions to voxel spacing, current voxels coordinates intensity, image size, added methods to OrthogonalView and ViewSeg to get the necessary information for the dialog)
Week 8 (5/12/011 - 11/12/11)	
2 hours	Merging git
1 hour	ImageDetails (function for displaying World Coordinates)
4 hours	Created classes for the tool menus (zoom, paintbrush, eraser, polygon segmented thresholds, connected 2d)
2 hours	Mainwindow (added QStackedWidget for displaying the menus for the tools one at a time depending on which tool is chosen)

Savvas	
Week 1 (17/10/11 - 23/10/11)	
Week 2 (24/10/11 - 30/10/11)	
4 hours	Setup Build Environment
2 hours	Overlaying lines (Investigated using a transparent QWidget)
Week 3 (31/10/11 - 6/11/11)	
4 hours	Setup Build Environment
4 hours	Overlaying lines (Drawing to a transparent QWidget)
3 hours	Merge (Tried to overlay lines in main project)
Week 4 (7/11/11 - 13/11/11)	
4 hours	Report 2
4 hours	Pen Tool (tried using built-in class vtkImageTracerWidget)
Week 5 (14/11/11 - 20/11/11)	
4 hours	Pen Tool (tried using built-in class vtkImageTracerWidget)
1 hour	Cursor Tool (tried using built-in class vtkImageTracerWidget)
3 hour	Cursor Tool (using VTK actors)
Week 6 (21/11/11 - 27/11/11)	
4 hours	Overlaying lines (Researching coordinate systems in VTK)
5 hours	Cursor Tool (Displayed in all three orthogonal views correctly)
Week 7 (28/11/11 - 4/12/11)	
4 hours	Pen Tool (Devised algorithm to make drawing smooth)
7 hours	Line Tool ()
3 hours	Line Tool (Researched algorithms and implemented one to fill the area defined by Line Tool)
Week 8 (5/12/011 - 11/12/11)	
2 hours	Line Tool (fixed a bug on the display of the lines)
2 hours	Line Tool (added curvy lines option to the tool)
1 hour	Added code to some buttons
2 hour	Pen Tool (Added shape functionality)
5 hours	Seed Tool (Implemented a visual part of the tool)
2 hours	Line Tool (Improved the auto-close feature of the tool)
2 hours	Line Tool (Research on the LiveWire algorithm)
3 hours	Line Tool (Implementation of the LiveWire algorithm through itk implementation online)
1 hour	Pen Tool (Connected some code to the main window buttons)
2 hours	Report 3

Shaan	
Week 1 (17/10/11 - 23/10/11)	
2 Hours	Setup Build Environment (Installed VS2010, CMake. Built VTK. Tried compiling example VTK projects.)
Week 2 (24/10/11 - 30/10/11)	
3 Hours	First Report
3 Hours	Setup Build Environment (Rebuilt VTK correctly)
5 Hours	Setup Build Environment (Made simple CMake project that used VTK and ITK libraries)
Week 3 (31/10/11 - 6/11/11)	
3 Hours	Opening Files (Loading a DICOM image)
3 Hours	Merging (Basic CMake project with a QT GUI, opening files, segmentation)
Week 4 (7/11/11 - 13/11/11)	
2 Hours	Refactoring
3 Hours	Image Navigation (Setting position of the three views)
3 Hours	Image Navigation (Slider bars change current position, and are updated to reflect current position)
4 Hours	Second Report
Week 5 (14/11/11 - 20/11/11)	
1 Hour	Orthogonal Views (Researched how to draw a cursor)
3 Hours	Opening Images (DICOM)
4 Hours	Display Image Bug (An orthogonal view displayed incorrectly: appeared flipped)
Week 6 (21/11/11 - 27/11/11)	
1 Hour	Display Image Bug (Corrected the flipped orthogonal view by changing position of vtkCamera)
2 Hours	Merging (Cursor Tool)
2 Hours	Refactoring
6 Hours	3D View (Researched how to display the segmentation mask in 3D)
Week 7 (28/11/11 - 4/12/11)	
2 Hours	3DView (Zoom to Fit feature)
6 Hours	3DView (Position Selection)
1 Hour	Bug Fix (Cursor didn't show in Linux)
Week 8 (5/12/11 - 11/12/11)	
3 Hours	Volume Calculation
5 Hours	Report 3
2 Hours	GUI Work (Orthogonal View buttons for the Line Tool)
2 Hours	Merging (Tool buttons appear only when a tool is selected)

Daniel	
Week 1 (17/10/11 - 23/10/11)	
2 Hours	Set up build environment on Windows 7 64-bit (Compiled VTK/ITK & Qt)
Week 2 (24/10/11 - 30/10/11)	
2 Hours	Researched CMake/CTest and Continuous Integration servers
1 Hour	Report 1
3 Hours	Set up build environment on Windows 7 64-bit (Enabled some additional features within VTK, linked VTK/ITK & Qt with SIMI)
Week 3 (31/10/11 - 6/11/11)	
4 Hours	Set up build & test environment on TS08 (Windows) and Linux lab machines
2 Hours	Researched C++ unit testing frameworks, and evaluated GTest
2 Hours	Patched CDash to make it compatible with DoC web servers
1 Hour	Added code to automatically create a mask image of the same dimensions as main image on image load
Week 4 (7/11/11 - 13/11/11)	
2 Hours	Created sample "Calculator" project, to test integration between GTest, CMake, CTest and CDash
1 Hour	Report 2
3 Hours	Fixed UI Bug in Windows (Open dialog caused crash due to string formatting on Windows)
Week 5 (14/11/11 - 20/11/11)	
2 Hours	Added feature to Save Mask Image
3 Hours	Added first unit test, and integrated CTest/CDash into out CMakeLists.txt file
5 Hours	Created build/test scripts for Linux and Windows, and set them to run via cron/scheduled tasks
Week 6 (21/11/11 - 27/11/11)	
1 Hour	Fix buffer allocation problem (now zeros memory, to prevent random data appearing within image)
5 Hours	Created Checkerboard Test Image Generator, to test if SIMI's cursor handles spacing between voxels correctly
Week 7 (28/11/11 - 4/12/11)	
2 Hours	Bug fixing (old image would remain on 3D view after loading new image)
5 Hours	Added Open Recent Images feature
Week 8 (5/12/011 - 11/12/11)	
3 Hours	UI Fixes for Windows
2 Hours	Bug fixing (vtkImageViewer2 would not render on new UI layout)
3 Hours	Merged new UI layout
3 Hours	Evaluated QTestLib for UI testing
4 Hours	Report 3

Chapter 3

Conclusions

3.1 Meeting Key Requirements and Extensions

We set out with a number of key requirements for the project, all of which were completed:

- Be able to load medical images of type DICOM and VTK.
- Be able to segment images using a manual pen/eraser tool, by colouring in 2D slices.
- Allow the user to perform 2D region growing on a 2D slice.
- Integrate with ITK such that you can extract 2D slices of the 3D image, and send them to ITK for processing.
- Calculate the volume of a 3D segmented object by counting the number of voxels it contains.
- An auto-save feature. Periodically save any segmentations created.

We also completed some of our extensions:

- 3D region growing.
- Build project on Linux.
- Implement an alternative algorithm to perform segmentation: Livewire

There were a number of tasks which we had outlined as extensions which we were unable to complete due to time constraints:

- Allow user to perform multiple segmentations in one file, and set a colour and a label for each segmentation.
- Have a (bounded) undo facility that can undo the last n user actions.

We found that despite working over the Christmas holiday, we would not be able to complete these extensions. Instead, we concentrated on fixing the small bugs and making improvements suggested by our supervisor.

We were also able to complete some features that we found useful or occurred to us during the course of developing the project:

- Used an integration server to display our test results.
- Created custom mouse functionality to the viewer so it can pan around the image.

3.2 Future Extensions

If we were to be given additional time to work on the project, the first feature we would implement would be an Undo feature, as listed in the original extensions, which we weren't able to complete due to time constraints. We often found ourselves wishing for an undo feature, during the course of testing features related to changing the mask (such as the tools).

Although we were able to implement two segmentation algorithms (LiveWire for 2D segmentation, and Connected Threshold for 3D segmentation), it would have been beneficial to support another 3D segmentation algorithm, to allow the user to choose between more than one in an instance where an algorithm produces spurious results.

Bibliography

[1] Connected threshold segmentation of ventricles in the brain. `http://www.insight-journal.org/browse/publication/194`.

[2] Itk mailing list. `http://www.itk.org/pipermail/insight-users/2009-June/031128.html`.

[3] A livewire implementation in itk. `http://insight-journal.org/browse/publication/230`.

[4] Vtk smart pointers tutorial. `http://www.vtk.org/Wiki/VTK/Tutorials/SmartPointers`.

[5] *VTK Users*. Kitware. ISBN 1930934238.

Appendices

Appendix A

User Guide

SIMI

MANUAL

Adrian Clay

Daniel Breed

Raluca Bolovan

Shaan Malik

Savvas Kleanthous

Section 1 - Table of contents

Contents

Section 2 - Overview

2.1 About this document

2.1.1 Subject

This document represents the user manual for SIMI.

SIMI is an image editing tool, for medical 3-dimensional images.

2.1.2 Abstract

The focus of this manual is to explain the different functions and tools offered by SIMI's user interface.

2.1.3 Addressee of the document

This document is addressed to anyone who wants to use SIMI.

2.1.4 How to read this document

This is a reference document, and so it is not necessary to read from cover to cover

2.2 Introduction

The intention of the Structural Identification in Medical Imaging (SIMI) is to allow a clinician to perform segmentation on medical images.

The medical profession produce and use CAT and MRI scans to non-invasively inspect the body and diagnose problems, without recourse to explorative surgery.

Segmentation - the process of partitioning an image into segments of interest - is an area of Computer Vision that has application in the medical setting, as if an organ or some other anatomical feature can be extracted from such scans, then further analysis can be performed (calculating the volume of an organ for example). Therefore, the ability to segment medical scans into anatomical features can aid in diagnosis.

SIMI provides functionality to traverse through the 3D dataset, via 3 orthogonal views, as well as display the result of applying a segmentation algorithm in 3D.

Section 3 - Working with SIMI

3.1 GUI Overview

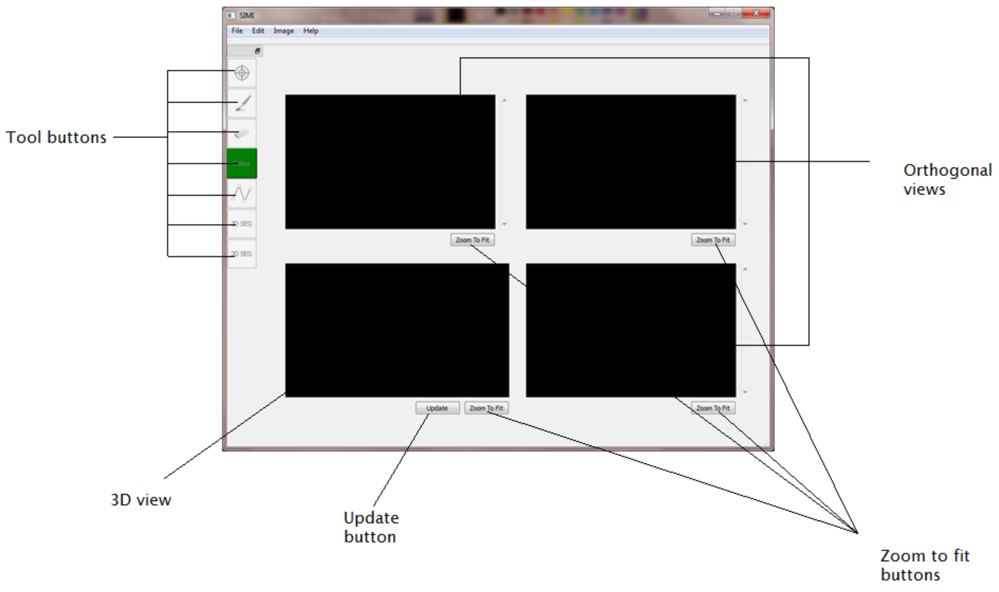


Figure 3.1 The main window as seen when SIMI is loaded

When one loads SIMI, the program opens up a window that looks like the picture above.

The three orthogonal views are going to display the 3D image, while the 3D view will display the 3D segment, after an algorithm has been applied and the Update button is pressed.

The Tool buttons will be individually explained in further detail in the following chapters and sections. Each of them represents a different functionality of the program.

3.2 Getting started

3.2.1 How to load an image

Let's get started now.

The first thing that you probably want to do is to load an image. SIMI supports DICOM, nii and vtk images. To load an image, choose the Open option from the File menu:

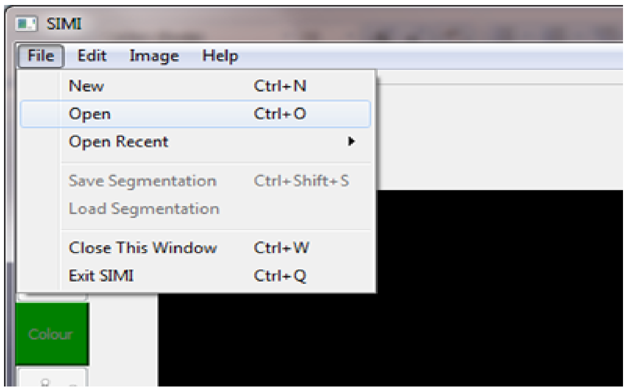


Figure 3.2 Open button

and navigate in your disk to select the picture you would like to load. When you click OK, and the image is loaded, the three orthogonal views, should have automatically updated. You should be able to see a crosshair, indicating the point that is currently selected:

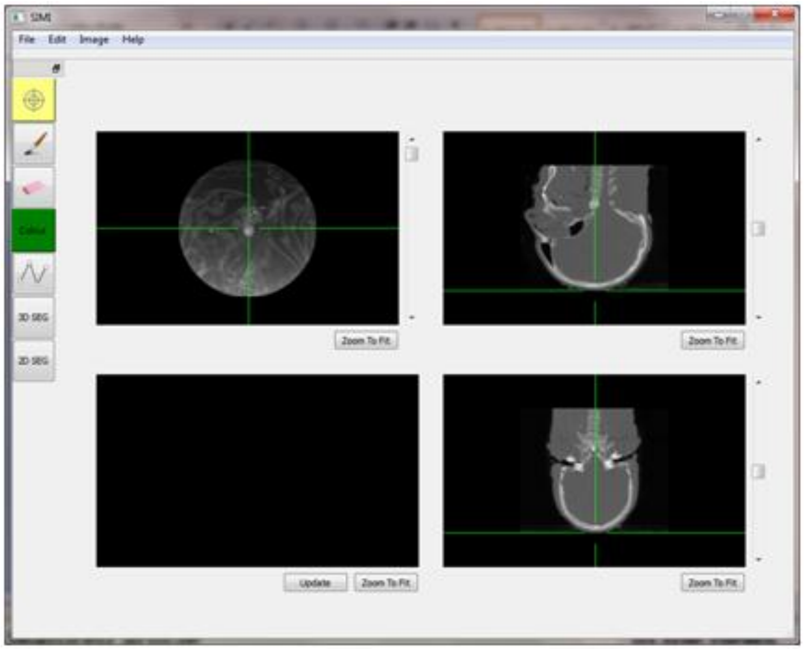


Figure 3.3 The main window as seen when an image is loaded in SIMI

From now on, you will be able to load the same picture in an easier way, using the Open Recent submenu:

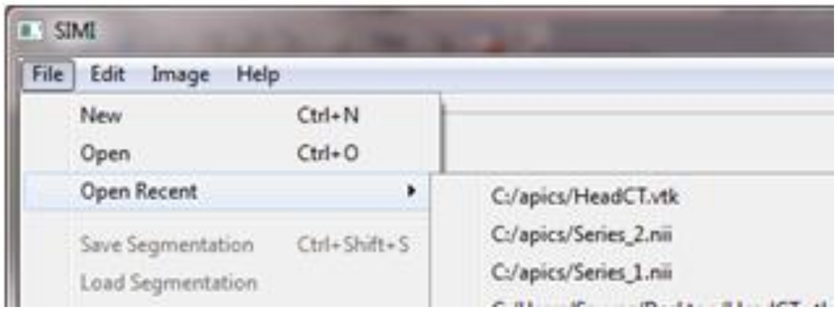


Figure 3.4 Open recent submenu

3.2.2 Orthogonal views

Before continuing on, you should first know a few things about the way the three orthogonal views work.

SIMI has an internal cursor that, at all times, focuses on some point of the 3-dimensional image. This point is defined by its three coordinates: the x-coordinate, the y-coordinate and the z-coordinate, all defined on a Cartesian 3-dimensional coordinate system.

Each of the three orthogonal views display a 2-dimensional slice of the image. An XY slice, given a z0-coordinate, is defined as the plane that is parallel to the plane defined by the X-axis and the Y-axis, whose points all have z0 for their z-coordinate.

The upper-left view displays the XY plane of the picture, the upper-right view displays the YZ plane and the lower-right view displays the XZ plane.

3.2.3 How to use the cursor

Having loaded an image, it is only natural that you might want to navigate through it. Fortunately the default tool that is selected by SIMI is the cursor tool. The crosshair that is visible on the three orthogonal views at all times, indicates the point that the cursor currently focuses on. But the crosshair can only indicate two of the three dimensions. The third dimension (for example the z-coordinate for the upper-left view) is indicated by the slider bars that are on the right of the three orthogonal views.

You can change the point of focus in two different ways.

The first is by manually moving the crosshair, when the cursor tool is selected, and the second is by re-adjusting one of the slider bars.

If you want to remain on the same image slice, but focus on another point, you only have to left-click on the point you want to focus on. Additionally, you may drag the crosshair by left-clicking and holding while moving the mouse. Dragging the crosshair is a great help when greater accuracy is required.

If you need to change the image slice you are viewing, you can move the slider bar. The uppermost position of the sliders represent the zero depth for a particular view, while moving the slider downwards increases the depth.

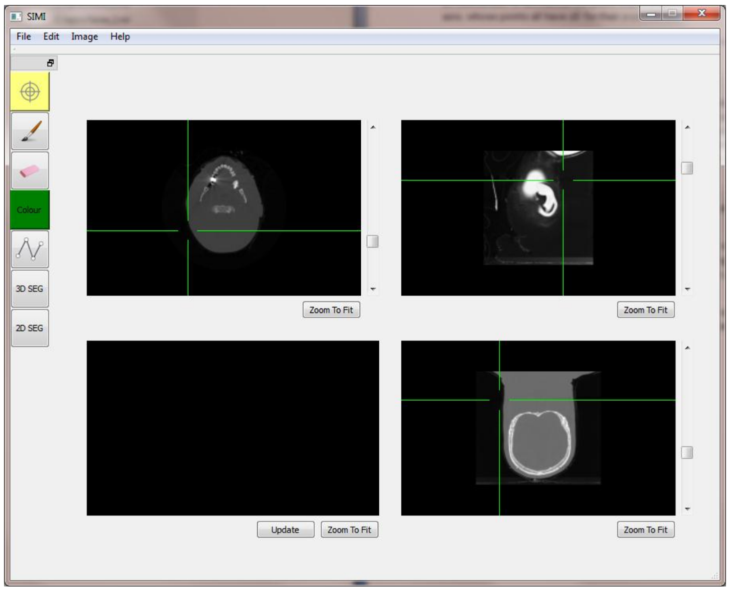


Figure 3.5 The main window after the cursor has been moved

3.2.4 Getting some extra details

When great accuracy is a concern, a simple crosshair might not do. You might also need to know the exact dimensions of the image you are working with.

Some extra details about the picture itself and the cursor can be found by choosing the Image Details option of the Image menu:

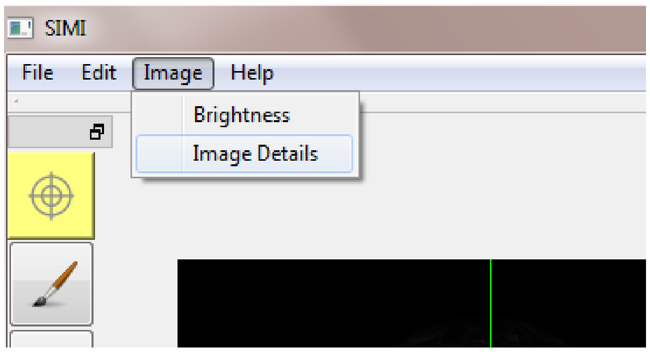
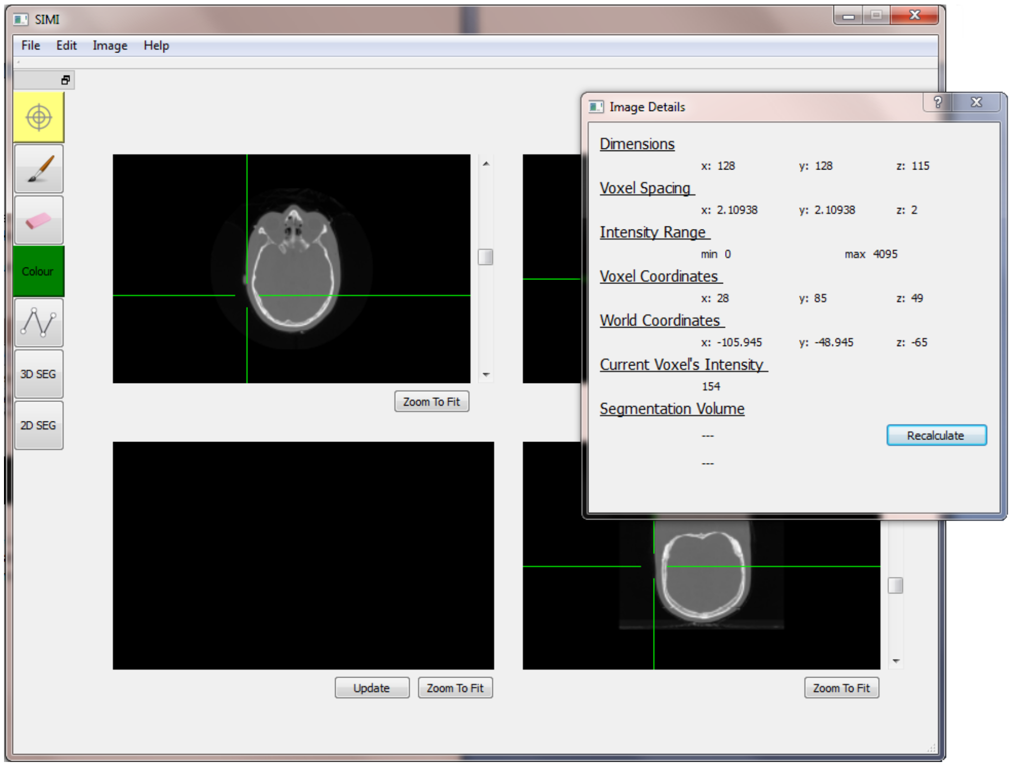


Figure 3.6 Image Details button

This should open up a dialog box with quite some more details about the picture and the current position of the cursor. Mind, that the information given by the Image Details window change dynamically. That means that moving the crosshair or the slider bar updates the given information. Additionally, you may calculate the volume of a segment (when you have started defining one) by pressing on the Recalculate button:



Dimensions
x: 128 y: 128 z: 115
Voxel Spacing
x: 2.10938 y: 2.10938 z: 2
Intensity Range
min 0 max 4095
Voxel Coordinates
x: 28 y: 85 z: 49
World Coordinates
x: -105.945 y: -48.945 z: -65
Current Voxel's Intensity
154
Segmentation Volume

Figure 3.7 Image Details pop-up window

3.2.5 How to zoom in and out

Zooming in and out in SIMI is extremely easy and flexible. All you need to zoom in a view is to roll your scroll wheel upwards while pointing to the view. As you might imagine, to zoom out, you will have to roll downwards your scroll wheel. This functionality is available at all times while an image is loaded.

If you wish to restore the initial state, all you need to do is press the Zoom To Fit button that is located under each view on the right.

3.2.6 How to drag the image

If you tried to zoom in the image, you might have noticed that the "camera" zooms in the centre of the slice. This is not very practical, as it limits the area of the image you may zoom in, as you cannot see other parts of the slice.

That is why the cursor tool has an image dragging functionality too: If the cursor tool is the selected tool, then right-clicking the image and holding, while moving your mouse, will move the whole slice.

Clicking on the Move To Fit button, will restore to the initial state any dragging you might have done on the slice.

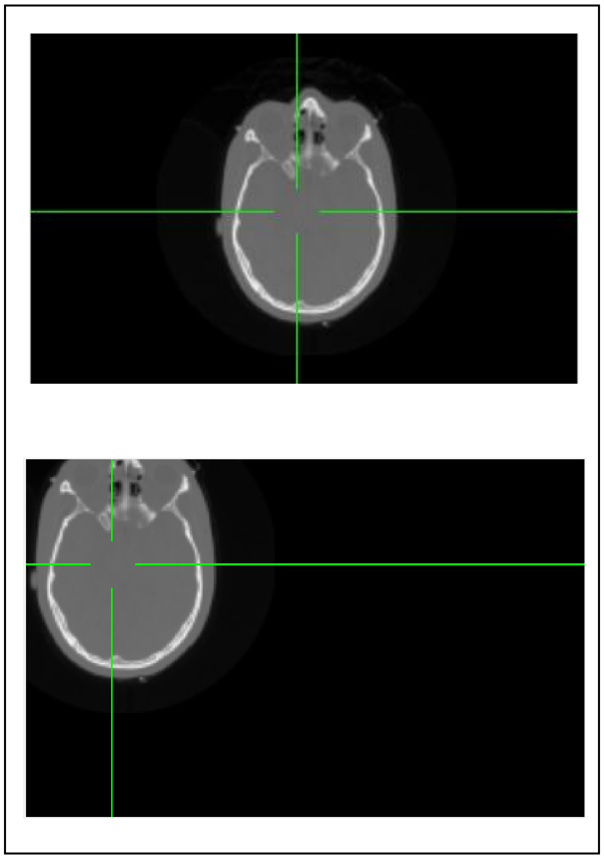


Figure 3.8 An orthogonal view as seen before and after some dragging

3.2.7 Adjusting the picture brightness

At any time during your work, you may adjust the minimum and the maximum brightness of the image display.

To do that, you will have to choose the Brightness option from the Image menu:

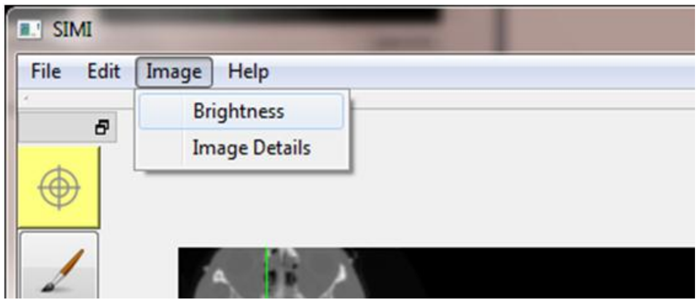


Figure 3.9 Brightness button

When you do that, a window will pop up.

On that window, you may re-adjust the minimum and the maximum brightness values by moving the sliders, as you wish:

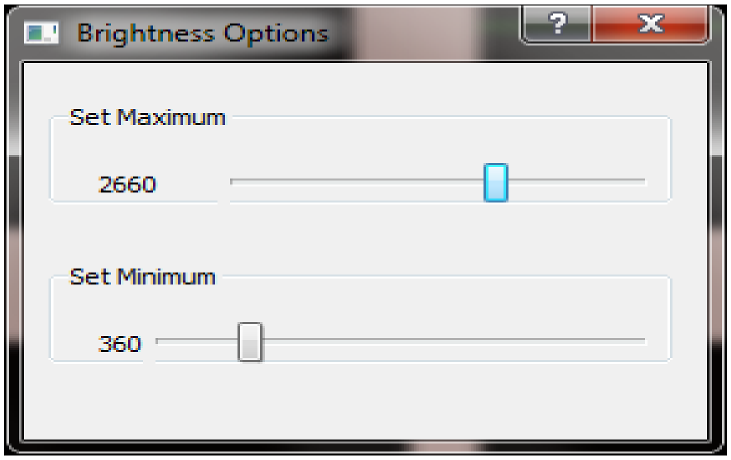


Figure 3,10 Brightness pop-up window

3.2.8 Exiting SIMI

To exit SIMI, you may use the Exit Simi option from the File menu. If you have more than one SIMI windows open, Close This Window will only close the current window, and keep the rest. Close This Window, will have the same effect as Exit Simi, if you only have one window of SIMI open.

Section 4 - Editing tools

4.1 The concept of the mask

4.1.1 What is the mask

The previous chapter have shown all the different ways you can use to navigate through your 3-dimensional image. That is all very good, but in most cases you probably need to make some observations on the image. You might want to manually or algorithmically select a part of the 3D image and inspect it, calculate its volume etc.

That is where the concept of the "mask" comes in. You can imagine the mask to be a 3-dimensional transparency where you can "note down" the segments of the image you might want to inspect. When you have made all your "notes" on the mask, you may inspect the volume on the 3D view, by clicking on the Update button that is located below the 3D view on the right.

4.1.2 How to edit the mask

There are two main ways to annotate your image's mask. You can either do it manually, or algorithmically by one of the algorithms that SIMI provides. All the different ways are going to be expanded in detail in the following sections.

SIMI offers two different tools that can be used to manually annotate the mask. The first one is the Drawing Tool that can be used as a pen on the mask. The other one is the Line Tool that can be used to fill an area defined by a set of lines.

Respectively SIMI offers two different algorithmic annotation methods. Those are the 2-dimentional segmentation and the 3-dimentional segmentation. Both of them will be explained in detail.

At any point, you may erase any part of your "notes" on the mask. This can be done by using the Eraser Tool. The Eraser Tool works exactly like the Drawing Tool, and you can think of it as an "inverse pen". For simplicity, the Drawing and the Eraser Tool, are going to be explained in the same section, as the way you can use them is identical. The only difference is the result.

4.2 Drawing and Eraser Tool

4.2.1 How to draw on and erase the mask

As mentioned above, the Drawing and the Eraser Tools are going to be explained together, as they have identical user interfaces.

To draw on the mask using the Drawing Tool, you will first have to select the drawing tool. To do that, you will have to click on the button with this image:

When you select the tool, its button should be highlighted with a yellow shade.

To draw on the mask using the Drawing Tool , just left-click on one of the orthogonal views and drag your mouse without releasing.

You are going to see a green semi-transparent line appear on the path of your motion. This indicates a note on your mask. You can draw whatever you wish to select on any view or slice of the image.

You may see what you have drawn on the 3D view by pressing the Update button:

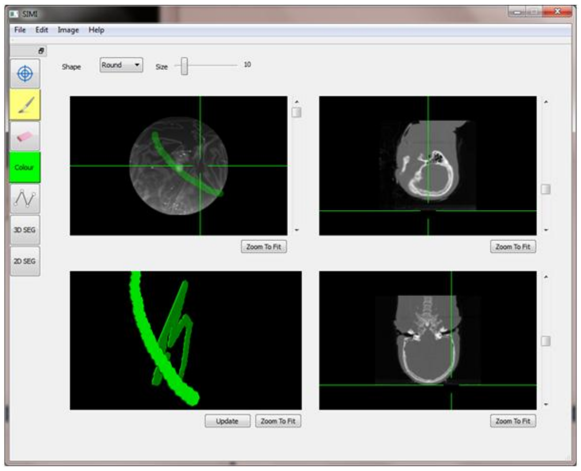


Figure 4.1 The Drawing tool in use

You may not be completely satisfied with the result you got. That is fine since we have the Eraser Tool. You may use the eraser in exactly the same way, to erase your notes. To select the Easer Tool, you will have to click on the button with this image:

Once the button is highlighted, you may use the tool:

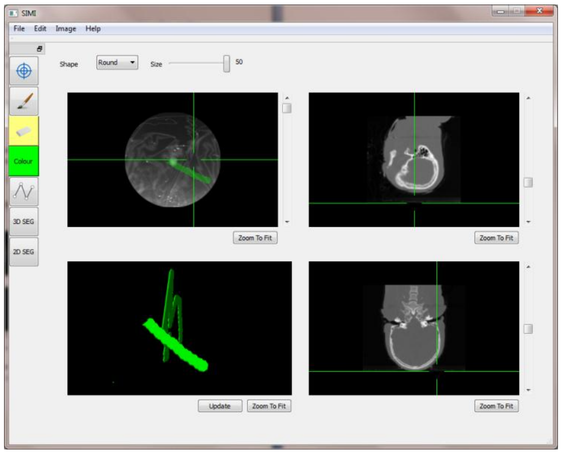


Figure 4.2 The Eraser tool in use on the previous image

4.2.2 Changing the settings of the Drawing and the Eraser Tools

There are two parameters you may adjust when using the Drawing or the Eraser Tool. The first is the shape and the second is the width of your brush.

SIMI allows you to draw with a round or square brush and change the width:

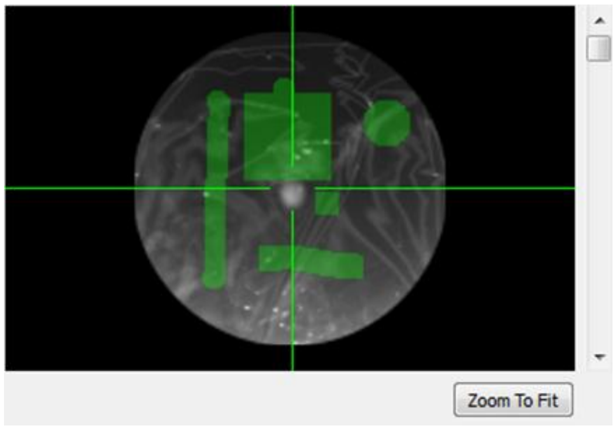


Figure 4.3 The different settings of the Drawing tool in use

To change the shape, just use the drop-down list that is located right above the upper-left view, and to change the size, just slide the bar that is located right next to the drop-down list.

4.3 Line Tool

4.3.1 How to use the Line Tool

The Drawing Tool that was described in the above section, is quite nice and flexible, but it is quite tiring when you would like to manually select large areas of interest. The Line Tool helps to cover this need. Using the Line Tool, you can select a region of a slice and with the press of a button annotate the whole area on the mask.

Let's get started.

First you will need to select the Line Tool by pressing on the button with the image:

When the button is highlighted, try clicking on one of the views. This will print a small red cross on the view you have clicked:

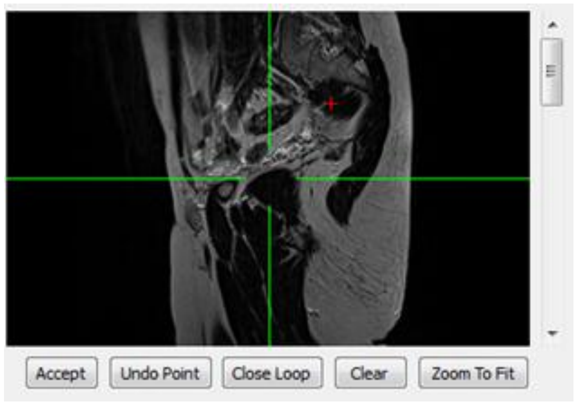


Figure 4.4 A view with one handle

This marks the first handle you have chosen. The Line Tools works by connecting handles on the image with lines. If you click again at another point in the same view, then the red cross will move to the point you last clicked and a purple line will appear between the two points:

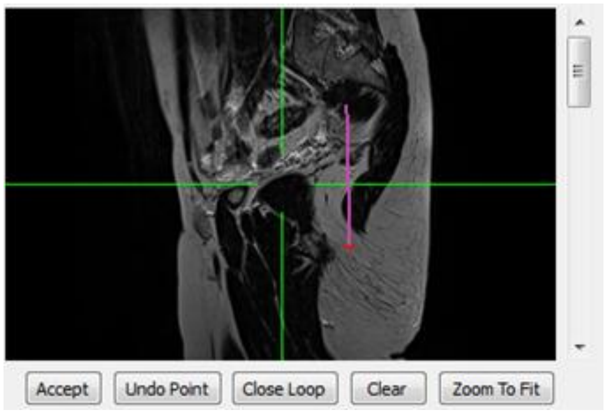


Figure 4.5 A view with a second handle added

You can add more handles -and lines- in the same way.

At any point, clicking on the Undo Point button, will remove your last handle, and the line connecting it to your penultimate one.

When you add a third point, you will see that a red dashed line that connects your first and last handles appears:

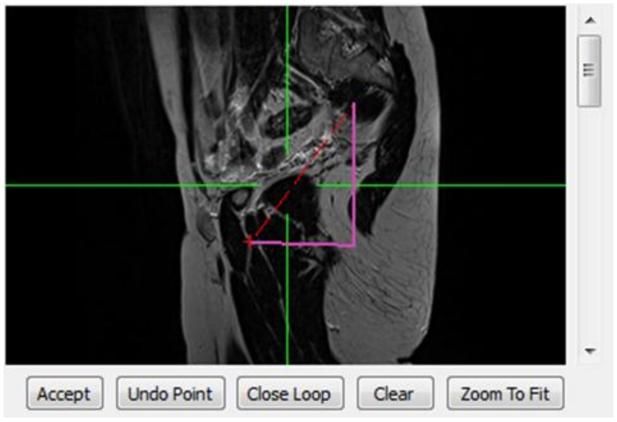


Figure 4.6 A view with a third handle added

When the red dashed line is present, it means that your handles may define a region. Your handles define a region if and only if their number is greater than two. When a region can be defined, the Close Loop button, that is located below the view, can be used.

Pressing the Close Loop button connects your first and your last handles. And paints all the lines red. This is to indicate that you have defined a region and that the Accept button can be used:

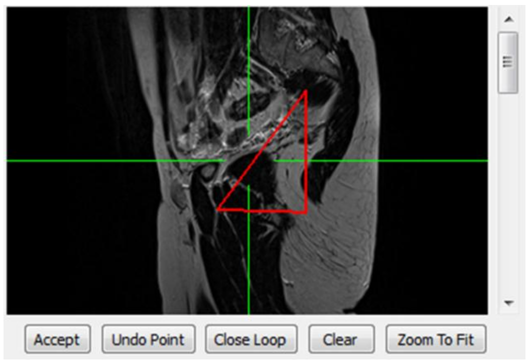


Figure 4.7 A view with a closed loop of lines

When your lines have turned red, then you can use the Accept button. This clears all your lines and "paints" this whole area on the mask:

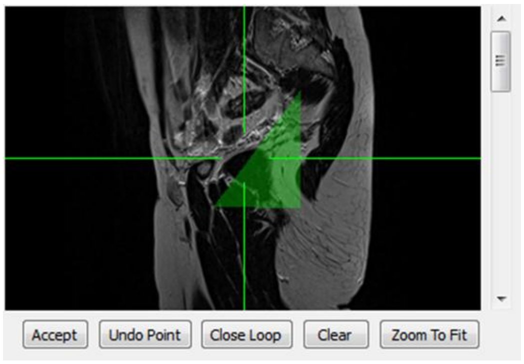


Figure 4.8 The result of pressing Accept on the previous figure's view

At any point, the Clear button, just clears all your lines and handles.

4.3.2 Linear and Curve mode

The Line Tool has three modes. The Linear mode, the Curve mode and the Livewire mode. This subsection will talk about the differences of the Linear and the Curve modes.

Let's talk about the Linear mode for now. This is the default mode of the Line Tool. You have already seen (in the previous subsection) how to use this mode. One thing that hasn't been covered is what happens when you try to drag. Left clicking and moving your mouse while on the Linear mode of the Line Tool, will draw a continuous line. In reality it produces a large number of handles all located near each other, giving the impression of a free curvy line. But trying the Undo Point button for a few times, you can see the different handles.

The only difference between the Curve mode and the Linear mode, is that when you press the Undo Point after you've produced a curvy line by dragging your mouse, in Curve mode, then the whole line will disappear. i.e. Only a single handle is produced when you drag your mouse.

4.3.3 Livewire mode

The third mode of the Line Tool is the Livewire mode. The Livewire, using an algorithm, will consider the colour differences in an image, and will try to predict the path you intend to follow. The way it works, it that it tries to follow a path where the colours change steeply.

You can use the Livewire in the same way that you use the Linear mode. Just add a new handle. Instead of connecting the points with a straight line, it will connect them with the livewire:

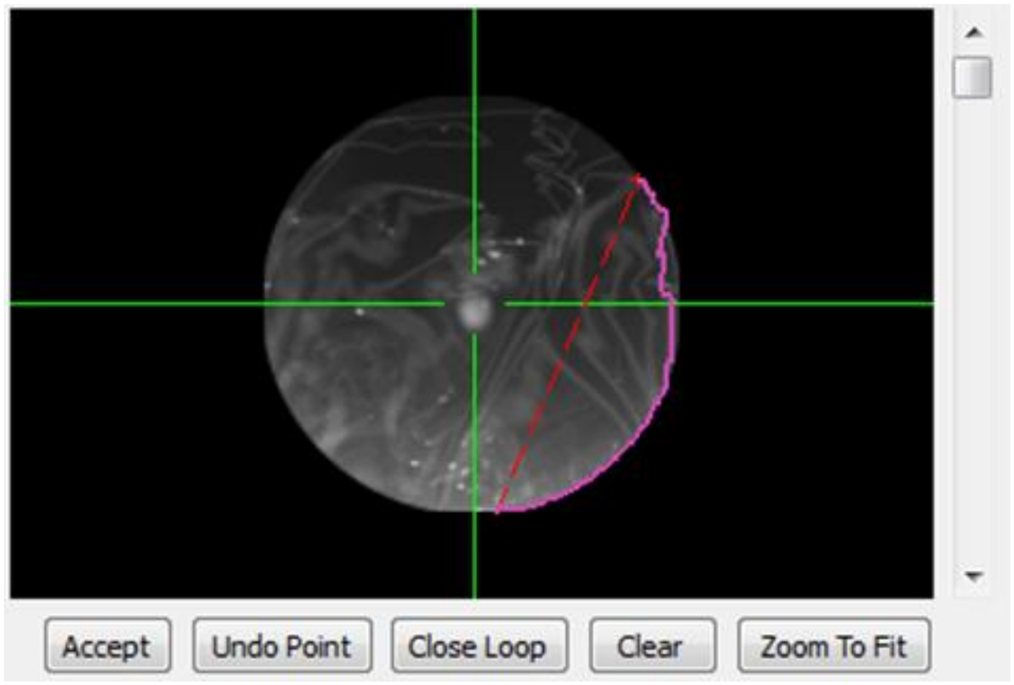


Figure 4.9 The livewire used with only one handle (other than the first one)

4.4 2D Connected Threshold Segmentation

SIMI provides semi automatic segmentation in the form of a connected threshold algorithm. It works in both 2D and 3D mode. In 2D mode the algorithm will only work in the slice that you select. To use the tool first load your 3D image as described in section 3.2.1, then select the "2D" button on the left hand side of the main interface. Once the segmentation menu at the top of the window is visible then select a seed point by clicking on any point in the image in any of the views. By adjusting the sliders in the segmentation menu you can change the lower and upper thresholds of the region growing algorithm. The result of the algorithm will be displayed as you change the sliders. Once happy with your thresholds, you can select a new seed point and perform the process again.

4.5 3D Connected Threshold Segmentation

The process for 3D connected threshold segmentation is similar to the process outlined above. First load an image and then select the "3D" button on the left hand side of the main interface. Then place the seed point for your segmentation in a similar manner and select the thresholds you wish for upper and lower. Once you have the correct thresholds selected click the "Compute" button to perform the algorithm and display the result.