Imperial College London

Department of Computing

Browser-based Medical Image Viewer using WebGL

by

David Basalla

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

September, 2014

Contents

1 Introduction

Medical imaging is a type of scientific visualisation that deals with the analysis, representation and exploration of medical image data. It should aid in diagnosis, treatment planning, support in actual operations, documentation and educational purposes. Medical imaging is primarily based on 3D volume data acquired by scanning devices such as computed tomography and magnetic resonance imaging. The resulting data files are viewed with specific computer software and contain a wealth of data. This software traditionally needs to be installed on a local computer machine. With the rise of *WebGL*, a powerful 3D Graphics API for the Web that makes use of the local graphics card, this project will investigate the general plausibility and specific implementation of a web browser based medical image viewer. This should free the user from having to use desktop bound software and instead enable him to simply log on to a website in his preferred web browser.

1.1 Motivations

To view medical image data, users are required to have appropriate software installed on their local desktop computer, which may involve a lengthy download or installation process. In this model software developers have to cater for different operating systems when writing these programs. When they have finished an update, it will again have to be delivered to the user (usually via download) and installed. To circumvent these issues, the goal of this project is to provide a medical image viewer that runs in any web browser. This web-app will aim to provide the same functionality as a desktop solution without being tied to a specific operating system. It furthermore aims to provide an intuitive and complete set of tools for viewing multiple files and allowing for creation, viewing and manipulation of label maps and other methods of annotating these images. The rationale behind this is to facilitate users to conveniently view, edit and annotate medical image files independently of location or type of computer, following the "Software as a Service" paradigm.

1.2 Requirements

The goal of this project is to build a web browser based medical image viewer. This will be achieved by making use of the modern graphics capabilities of modern web browsers such as *Canvas* and *WebGL* to display 2D and 3D graphics. In general the project aims to provide similar functionality and user experience as a comparable desktop application (which will be discussed in the next section). The essential requirements that the resulting application should meet are as follows:

- The user can load and view a scan file (of type *NIfTI*)
- The user can view a file in 2D and 3D
- The user can easily navigate through the 2D and 3D viewers, with emphasis on intuitive user experience
- The web-app provides controls for viewing the scan files at different brightness and contrast levels
- The user can load and view a labelmap for a given scan file
- The user can load additional scan or label maps into the viewer and compare them to previously loaded files with a set of intuitive controls
- Provide different colortables (for labelmaps, heat maps)
- Provide sample data to use if user has no files of his own

Given enough time, secondary requirements can be implemented, which are considered to be more complicated:

- The user can paint a custom labelmap for a given scan file with a set of intuitive tools. The primary goal will be that the user can paint areas manually on individual slices. Provided there is enough time, some semi-automatic painting tools will also be implemented that will allow the user to specify regions in the 3D space with automatic selection tools.
- The user can save a labelmap to his computer
- The user can create and save custom annotation to label given scans
- The web app provides a method for sharing data across the internet
- The web app provides image filters that can be applied to a scan or label map
- The web app provides measuring tools that can be applied to a scan or label map
- A three dimensional, parameterised model of the scan be viewed in the 3D view
- The 'cine mode' feature allows for an animation along a specified axis

1.3 Use Cases

A small selection of use cases have been discussed with the project supervisor. The following were suggested:

- A user loads two medical images files from a patient that has undergone brain surgery. The user wants to compare the scans from before the operation to afterwards. The user needs to be able to compare the two files in meaningful ways, with tools for image comparison and compositing.
- A user wants to create a custom label map by highlighting a certain section of a loaded scan. The user can use provided paint tools to paint individually on each slice, or alternatively use semi-automatic tools like filling in regions within a given color range to speed up this process. The user can save out this custom label map in *NIfTI* format.
- The user wants to create annotations for a file. He can use the software to create annotated planar or cuboid regions to for example specify where a given organ is in the scan. The software provides intuitive controls and helpful visual feedback to facilitate this process. The user can save out these files as a custom data file.
- A user wants to communicate with another user who is not present in the same location. The first user can use the software to provide links to specified medical image files to the second user. The second user can load these files with the software for analysis.

1.4 Contributions

The final software named *ScanView* represents a comprehensive application for viewing medical image data as well as adding custom annotations. All of the primary aims as stated above have been met, as well as a couple of secondary requirements (see full feature list in the Implementation Section). It is designed with intuitive input controls as advised by relevant literature for use by radiologists, mimicking conventional controls in other software. Judging by similar web applications online, *ScanView* provides many more features than its competition. It has also been tested successfully to run across a selection of different operating systems and web browsers, although it is not usable in every setup yet. The project was built on the *XTK* library, which appears to be the most comprehensive library for viewing medical image data to date. However since the library missing a lot of critical functionality as required by this project, the library had to be extended to provide tools for more dynamic application building. The final product demonstrates that with recent advances in web technology it is possible to create a viable browser-based alternative to desktop-based medical image viewing applications.

2 Background

2.1 Medical Imaging

As mentioned, medical image files are acquired by scanning devices such as CT, MRI or PET scanners that output data at a given resolution of data points. A typical medical image file contains a stack of individual images. Each image represents a thin slice of the scanned subject and is made up of pixels. For a volume data set the stack of images can be usually subdivided into cross sections along the three Cartesian axes X, Y and Z and together form a 3D Grid. In medical terms, these orthogonal views are called axial, sagittal and coronal. A constant pixel distance along X and Y allows for accurate measuring of distances and areas along a slice. The slice distance is the measurement in space from one slice to the next along its cross section. The three distances in every direction combined make up the voxel distance. If the pixel distance is equal to the voxel distance, the dataset is called isotropic. However in practice datasets usually exhibit a much smaller pixel distance than slice distance, which is called anisotropic. Measuring cross-sectional areas and volumes are valuable for example in diagnosis of vascular diseases. The quality of these measurements is highly dependent on the quality of the image data. Typically diagnosis has been performed these slices one at a time, meaning looking at every slice. These views support precise exploration and analysis and it is a legal requirement for radiologists to inspect every slice. However for spatially complex cases a purely slice-based presentation might not be very helpful. In this case 3D visualisation or volume rendering gives a better overview. The two different visualisations can be helpful for different users, such that radiologists would rely more on the 2D Views and physicians who carry out interventions might find benefit from 3D visualisations. Therefore any software that aims to display such medical image data should provide both views.

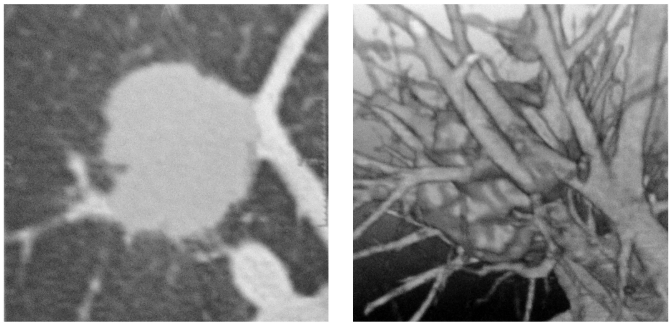


Figure 1: 2D View (left) and 3D View (right) (Preim & Botha, 2014)

2.2 Guidelines for Medical Image Viewer Software

As defined in 'Visual Computing in Medicine' (Preim & Botha, 2014) [1], a program needs to provide the following functionality in order to be of use when displaying this kind of medical imaging data.

First, a certain mapping of data to gray values has to occur, called 'windowing'. This is customisable, but the goal is to display an image with meaningful values so that a user can visually perceive the files. Different colortable mappings or lookup tables (LUTs) can be applied to render the same data with different color or level ranges.

Secondly, the user should be able to easily browse through the slices. The axial, sagittal and coronal views should be provided as well as a 3D dimensional view of these slices in either card-based or volume form. All slices should update accordingly as the user browses through them. However in order to examine a view in detail, it should be possible to enlarge a desired view. Displaying all views at the same usually only serves to give an overview. Additionally a feature called 'cine mode' could be provided which is an animation through one of the slice stacks. This could aid in understanding the spatial relationship of the slices.

Furthermore, the software should provide means to take measurements of the image data, such as distance, area or volume measurements. In this case it is helpful to specify a region of interest

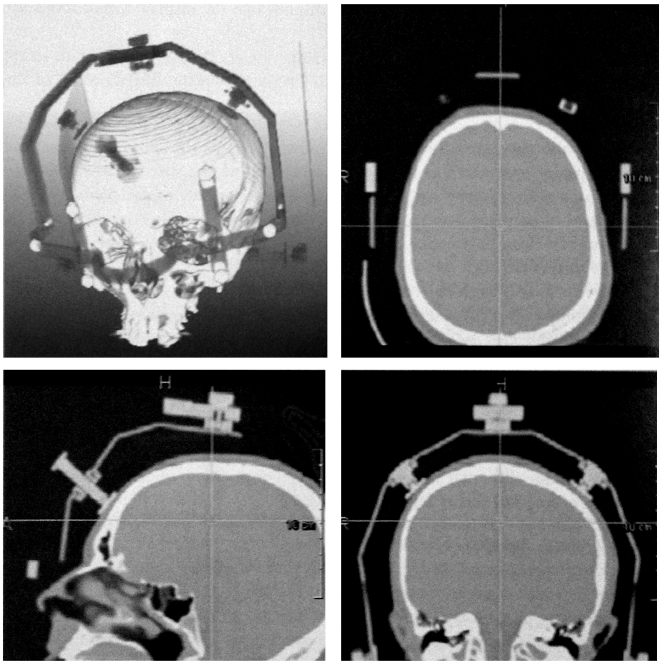


Figure 2: 3D View and Orthogonal Views (Preim & Botha, 2014)

(ROI) that can be focused on. Also intensity distribution can be measured to give indications of severity of disease in certain cases (such as osteoporosis) [1].

For advanced tools, the book suggests providing segmentation tools for identification and marking of certain structures. These tools could be manual so that the user has to paint a region for each slice, or semi-automatic where algorithms select regions following specific rules given by the user, such as threshold-based segmentation or region growing. The resulting segmentation (or labelmaps) should be overlaid on top of the scan data in a clear manner.

It is also suggested to have tools for annotating files with information. Examples would be creating a region around a specific organ or an arrow pointing to a particular area of interest. On the topic of user interaction, the book points out that the most frequently used should be carried out with the mouse. Specifically, browsing through slices could be done by scrolling wheel, right-clicking and moving the mouse affect the windowing, ie contrast and brightness. Zooming and panning should also be implemented in similar fashion. These kind of direct mouse controls are preferable to having to adjust specific sliders, although these may be included for visual feedback.

Additional functionality should be added for pressing down the Alt or Shift key and using the mouse. With regards to 3D rendering of the data, different modes are mentioned. Multiplanar Reformatting (MPR) is a way to creating an arbitrary slicing with an orientation that does not follow the cartesian axes. Maximum Intensity Projection (MIP) is a frequently technique which casts out rays from the viewing plane and highlights voxels with the maximum intensity, mostly used for diagnosis of vascular structures. Surface Shaded Display (SSD) is another visualisation technique that renders the data as 3D shaded surfaces based on a given threshold. This is achieved by connecting adjacent voxels as polygons. Some simple lighting is applied for shading so depth relations are more recognisable. Finally Volume Rendering produces a semi-transparent rendering of the data voxels where individual voxels' brightness determine their opacity. This is often referred to as direct volume rendering (DVR) as opposed to indirect volume rendering of SSD.

2.3 NIfTI File Format

After consulting with the project supervisor, it is deemed sufficient to support the *NIfTI* for format for medical image data. *NIfTI* supports the multi- dimensional format of medical image data as described in the previous section. The format is adapted from the widely used *ANALYZE 7.5* file format. The *ANALYZE 7.5* file format has been widely used in the functional neuroimaging

field. The files can be used to store voxel-based volumes. An *ANALYZE 7.5* data item consists of a file with the actual data in a binary format (with the filename extension .img) and another header file (header with filename extension .hdr) with information about the data such as voxel size and number of voxel in each dimension. Among other improvements, the *NIfTI* format allows for storing the two files in one file (with the filename extension .nii), which is an obvious advantage for file management. In the *NIfTI* format, the first three dimensions are reserved to define the three cartesian spatial dimensions (X, Y, Z). The fourth dimension is reserved to define the time points. The remaining dimensions, from fifth to seventh, are for other uses.

2.4 Desktop Applications

There exists a variety of desktop software to date that allow viewing of medical image data. For this project, a selection of programs have been tested and analysed for functionality.

2.4.1 MITK 3M3

MITK 3M3 by Mint Medical and German Cancer Research Center allows for easy viewing of a variety of medical image file formats. The layout of the views is customisable and by default shows the X, Y, Z dimension and a 3D view of all 3 dimensions together. There is a universal range slider that affects all images at once. Labelmaps (called Segmentations) can be added to a scan file, as sub layers and a range of paint tools are provided to colour in regions of the scan. An arbitrary number of layers can be created and saved out as separate files. Furthermore it supports 3D volume renderings, image filters and supplies measuring tools. In general it is a well rounded product with intuitive controls so will serve as a good benchmark test for the final product.

2.4.2 Imview

Imview is a medical image viewer created by Dr. Ben Glocker (who is also the project supervisor), which also allows the user to load scan files and inspect the individual slices of each axis. The user can perform flip transformations and apply different colortables to the image, such as Colormap JET which looks like a heat map. It also provides the option of displaying an information overlay with statistics about the image. It would be beneficial if the web-app has similar options as they provide useful data for the user.

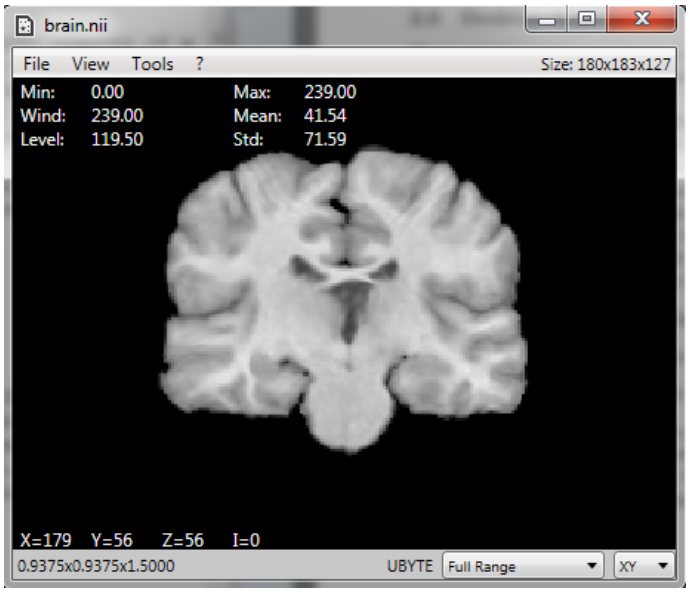


Figure 3: Screenshot of Imview User Interface

2.4.3 3DSlicer

3DSlicer is a free open source software that provides a modular platform for image analysis and visualization. Of all the packages, this tool seems to be the most feature rich, which is not surprising as it is funded by a number of American organisations such as the National Alliance for Medical Image Computing, Neuroimage Analysis Center and others. Although it features a wide variety of tools, the focus will be on the ones relevant to this project. *3DSlicer* allows for loading of different files of different formats. It provides options to composite different slices on top of each other for comparison. It supports hardware accelerated volumetric rendering of medical image data. It is widely customisable with global settings. Layouts are also customisable. It has an extensive set of tools for manual and automatic image segmentation. It has an in-built feature to download sample data, which would be a useful inclusion for the web app project. Another powerful feature is that *3DSlicer* allows the addition of new functionality through custom plug-ins and provides a number of generic features not available in competing tools. All in all, *3DSlicer* appears like a well designed, user-friendly program that has an overwhelming amount of functionality. It will be another good benchmark and inspiration for this project.

2.4.4 MIview

MIView was programmed by Greg Book, is open source and provides much the same functionality as *MITK*. *MIView* is an *OpenGL* based medical image viewer which aims to support a wide range of medical imaging files such as *NIfTI*, and raster images. The main goal of *MIView* is to provide a platform to load any type of medical image and be able to view and manipulate the image. Volume rendering is also supported. Control-wise, support for mouse-wheel scrolling seems erratic, which makes navigation through the slices more cumbersome. Also the buttons tend to be quite small which also does not improve the user experience. It does not appear to support the loading of multiple files together. A number of predefined color look-up tables are provided.

2.4.5 ITKSnap

Towards the end of this project, a survey was included in the web-application that asked users about which medical image viewers they used, if any. *ITKSnap* was mentioned the most frequently, so a brief review is included here. Development on this application started before the year 2000, is currently being led in development by Paul A. Yushkevich (University of Pennsylvania) and Guide Gerig (University of Utah) and is supported financially by the US National Institute of Biomedical Imaging and BioEngineering. It is currently in its third major revision.

Upon starting up the *ITKSnap* application, it is instantly appealing. It features a library of recently viewed files, which is very convenient for repeated use. It offers the ability to save a workspace as an *XML* file which includes which files are currently being viewed, making it easier to share specific sessions with other users. This would be a great addition to this project.

The controls follow the standard setup, with panning, zooming and traversing all assigned to the different mouse buttons. Scrolling through the indices is performed by scrolling with the middle mouse button. A slider is also provided on each panel for selecting a specific index. There are many options for navigating the slices, which seem well thought out and focus on convenience. A button is provided to center the image on a selected cursor position.

In terms of viewers, it offers zoomed-in views with a little thumbnail in the corner of the whole image to show which portion is being examined. When trying out some sample data, the 3D view did not display properly. Interestingly it does not feature many options for adjusting the layout of the viewers, with only a four-panel layout and a single panel layout available. As it has been rewritten with the UI graphics library *Qt*, these features should be very easy to implement.

ITKSnap further impresses with the customisation options for image brightness, contrast and colortable lookups. A wealth of options are available for each component, and a wide number of colortables are provided, which can also be edited. A separate information panel summarises the key data facts about the currently loaded files such as dimensions, voxel spacing and orientation.

The program als offers a variety of tools for segmentation (painting custom labelmaps), with manual and semi-automatic tools provided. It represents a volume of work that would be too ambitious to emulate in this project, as it only lasts three months.

In conclusion, *ITKSnap* appears to be the most appealing, focused and well rounded medical image viewing software out of the ones that were researched for this project, due to the wealth of customisation yet convenient design. It has a number of features that would make great additions to the current *ScanView* program. Unfortunately I came across this specific piece of software quite late so have not had the opportunity to add any of these features.

2.4.6 Photoshop

Looking further afield to some graphical editing software programs, some interesting functionality can be discovered. Adobe's *Photoshop* is well known software for creating and manipulating 2D imagery. When editing an image, the user works with a document. The user can add as many layers as required and edit each layer individually. The layers are composited to form the whole picture. Each layer's opacity and compositing/blending mode can be specified. This could be a worth while to emulate for comparing different medical image files in the web app.

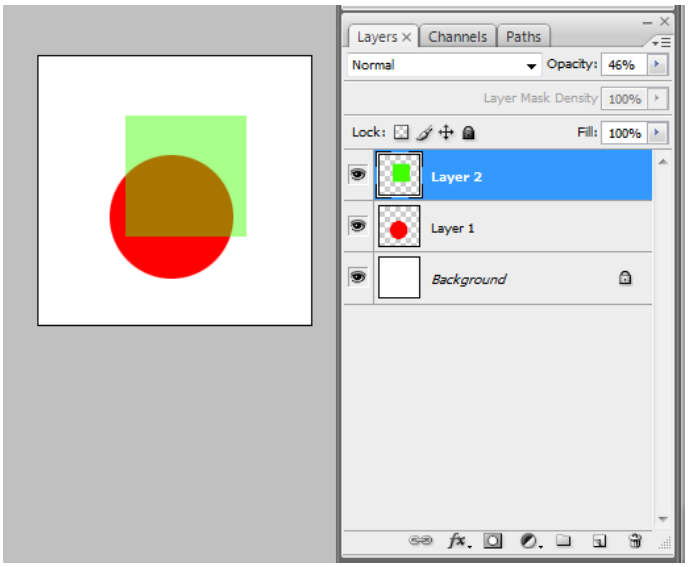


Figure 4: Layer Compositing in Photoshop

2.4.7 Nuke

The Foundry's *Nuke* is a compositing package which is used in the Visual Effects Industry to edit and composite 2D images with each other. It is used among others at high profile Visual Effects companies such as *Industrial Light and Magic* and *Weta Digital* [20] [21]). The interface differs from *Photoshop* in that it provides the user with a workplace where any number of nodes can be created. When selecting any node, a graphical display will appear. These nodes can be linked together and do anything from loading a specific image file, transforming the whole node tree or applying an image filter such a film grain. The power of this approach is the modular nature of a tree. Any node can be taken out and reinserted at another place into the tree, thereby changing the final image. Additionally, for comparing different images, by default the software supplies two image buffers which the user can fill with any image of his choice (including of course the result from any node tree in his workspace). Once both buffers are filled, the user can easily toggle between the two images, blend between them by setting the opacity and even use a slider to reveal just a portion of either image. So while *Nuke*'s node-based approach may be overkill for this project, *Nuke*'s image blending seems quite desirable for this project as it would give the user a variety of ways in which he can compare multiple images.

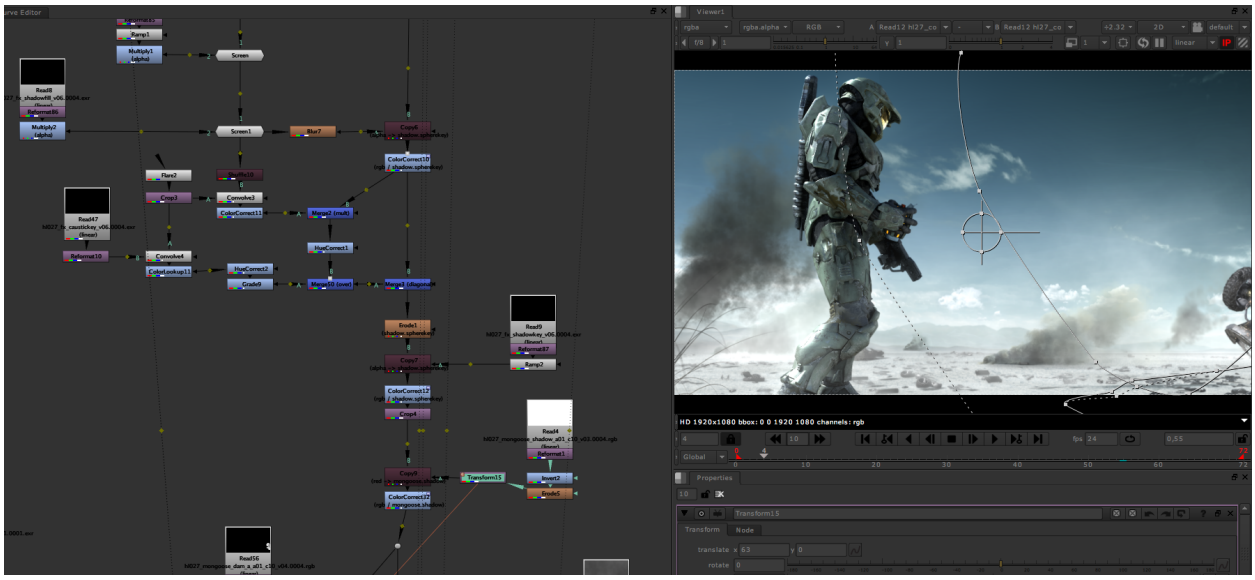


Figure 5: Nuke's node-based workspace

2.5 Developing Web Applications

2.5.1 Web browsers

According to web browser usage statistics from w3counter.com for July 2014, the four most used browsers are (in descending order) *Chrome*, *Internet Explorer*, *Firefox* and *Safari*. *Chrome* leads with 38.5% market share, followed by *Internet Explorer* (21.2%), *Safari* (15.4%), *Firefox* (15.2%) and *Opera* (3.1%) [24].

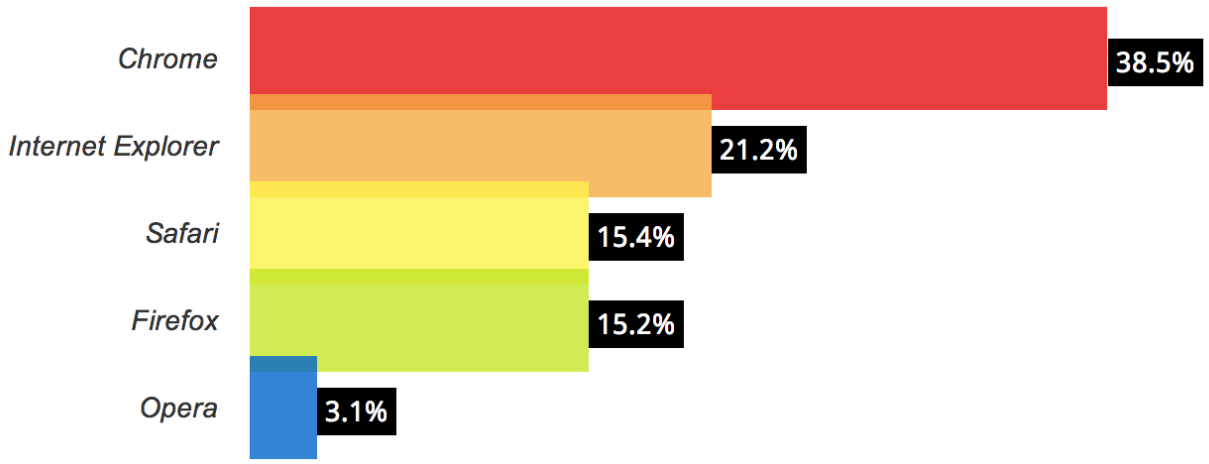


Figure 6: Browser Market Share July 2014 (Image taken from www.w3counter.com [24])

Looking at a breakdown over the last 10 years in Figure **??**, *Internet Explorer* used to be the most used web browser in 2007 with 67.6%, but its dominance has steadily declined up until the present day. In the same time span Mozilla's *Firefox* gained popularity and its market share peaked in 2010 but also decreased afterwards to equal *Safari*'s market share in July 2014. *Safari*'s market share has stayed relatively constant until it more than doubled in 2012 (possibly linked to the rise in iOS devices like Apple's IPhone and IPad around 2012 [23]). Google's *Chrome* web browser rose steadily from 2009 onwards and beat *Internet Explorer* in 2012 to become the most used web browser at present time.

The site www.w3schools.com, a popular tutorial site for web programming, has slightly different statistics. *Chrome* still comes out on top with 59.8%, but the places for *Firefox* and *Internet Explorer* are reversed with 24.9% and 8.5% market share respectively. W3School's sample set

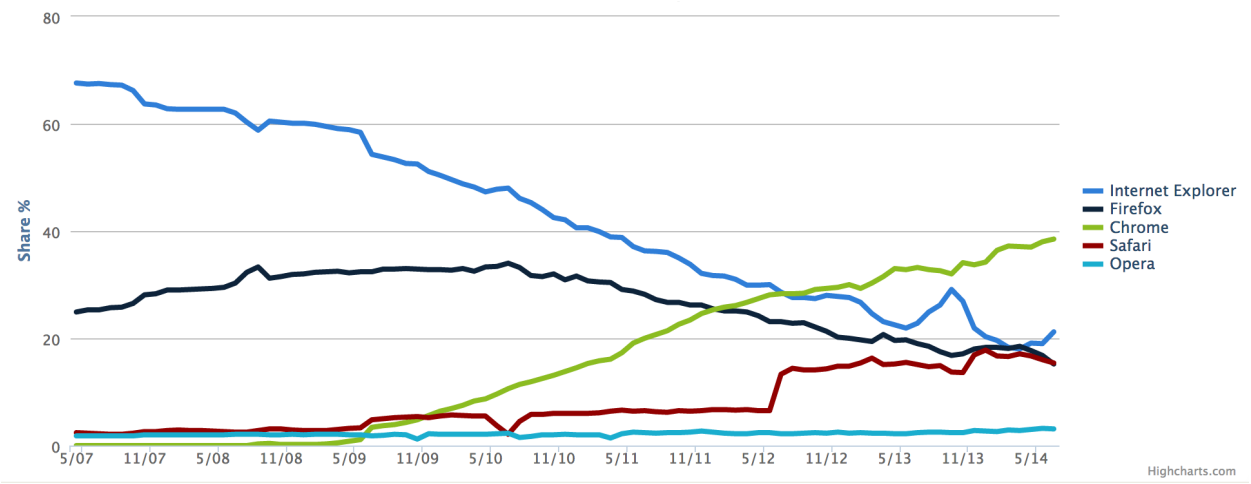


Figure 7: Browser Market Share History (Image taken from www.w3counter.com [**?**])

comes from people logging onto their sites, so therefore are more likely to be developers rather than casual users. This could mean that developers prefer *Chrome* and *Firefox* (a group where I would include myself).

Browser Statistics

2014	Chrome	Internet Explorer	Firefox	Safari	Opera
July	59.8 %	8.5 %	24.9 %	3.5 %	1.7 %
June	59.3 %	8.8 %	25.1 %	3.7 %	1.8 %
May	59.2 %	8.9 %	24.9 %	3.8 %	1.8 %
April	58.4 %	9.4 %	25.0 %	4.0 %	1.8 %
March	57.5 %	9.7 %	25.6 %	3.9 %	1.8 %
February	56.4 %	9.8 %	26.4 %	4.0 %	1.9 %
January	55.7 %	10.2 %	26.9 %	3.9 %	1.8 %

Figure 8: Browser Market Share 2014 (Image taken from www.w3schools.com [25])

Further statistics from w3c show that most web browser usage comes from *Windows 7* (39.48%) followed by *iOS 7* (10.10%) and *Windows XP* (9.64%). The most used screen resolution is 1366x768 pixels (19.27%).

Given the statistics discussed above, it was decided to mainly develop for Google's *Chrome*. The other browsers will be supported as secondary priority. It will be developed and tested on *Windows 7*, *OSX Mavericks* and *Linux Ubuntu*, which should cover a representative selection of Computer Operating Systems. In terms of screen resolution, the most widely-used screen resolution of 1366x768 will be taken into account, but a higher resolution assumed, such as 1920x1020. This is obviously an important issue, since a screen with HD resolution (1920 x 1080) and a screen with a smaller resolution of 1366x768 will need to have differently sized components. Therefore using absolute pixel values, such as setting a button or section to be exactly X pixels wide could cause problems, and care will have to be taken to make the website workable across different resolutions.

2.5.2 Developing websites

A web-based application is in essence a website that aims to provide a richer and more complex functionality than merely displaying text. At a very basic level, a website can be written with just *HTML* (Hyper Text Markup Language) code. *HTML* is written with tags as in the example below. This code is converted into a tree format of *JavaScript* node objects or Document Object

Operating Systems		
1	Windows 7	39.48%
2	iOS 7	10.10%
3	Windows XP	9.64%
4	Windows 8	8.90%
5	Android 4	8.57%
6	Mac OS X	6.42%
7	Windows Vista	4.21%
8	Linux	2.14%
9	Android	1.35%
10	iOS 6	0.97%

Screen Resolutions		
1	1366x768	19.27%
2	1024x768	11.56%
3	1920x1080	6.90%
4	1280x800	5.95%
5	768x1024	5.62%
6	1280x1024	5.04%
7	1440x900	4.65%
8	1600x900	4.43%
9	320x568	4.09%
10	320x480	2.85%

Figure 9: Most used Operating Systems (left) and Screen Resolutions (right) (Images taken from www.w3counter.com [24])

Model (*DOM*) by the web browser. The purpose of this is to provide a programmatic interface for scripting (removing, adding, replacing and modifying) this live document using *JavaScript* (Lindley, 2013) [4]. Different tags can be added to an *HTML* element, such as the *class* tag, which gets used to hook extra attributes into the element. These nodes can also be created by running *JavaScript* methods such as the *createElement()* function. The important point is that *JavaScript* and *HTML* are linked very closely, in that *HTML* can be programmed with *JavaScript*.

```
<html>
<body>
Hello World
</body>
</html>
```

Creating websites with just *HTML* however is not quite enough in today's world of modern web development. Modern websites can be thought of as a combination of structure, style and interactivity [26]. These three jobs are handled in turn by *HTML*, *CSS* and *JavaScript*, which go hand in hand to deliver modern web page content. *CSS* was first published and recommended by the World Wide Web Consortium (W3C), the main international standards organization for the World Wide Web, in 1996. It is generally used to style the look of *HTML* content, adding attributes such as font, color, dimensions and more many. This was previously possible with just *HTML*, but involved a lot of work as elements had to be type set individually and was still relatively limited compared to what is possible with *CSS* today. With *CSS* an *HTML* element can be assigned a class (via the *class* tag), which will refer it to specific set of *CSS* rules for setting style and look.

Additionally *JavaScript* can be included as code itself in an *HTML* document with the *script* tag. See the example below. *JavaScript* is used to add functionality to a website by managing interactive elements, function calls and whatever else a developer can think of. Advanced features such as *WebGL* and *HTML5 Canvas* are manipulated with*JavaScript*. When a website contains *JavaScript*, it is downloaded to the user's local machine and run locally ('client side') by the web browser's *JavaScript* interpreter. With increasing computational power of machines and faster processing speeds of *JavaScript* by web browsers, complex code (after it is downloaded to the local computer) can be run locally and relies only on the local hardware, not the network speed.

```
<script>
document.write("Hello, World!");
</script>
```

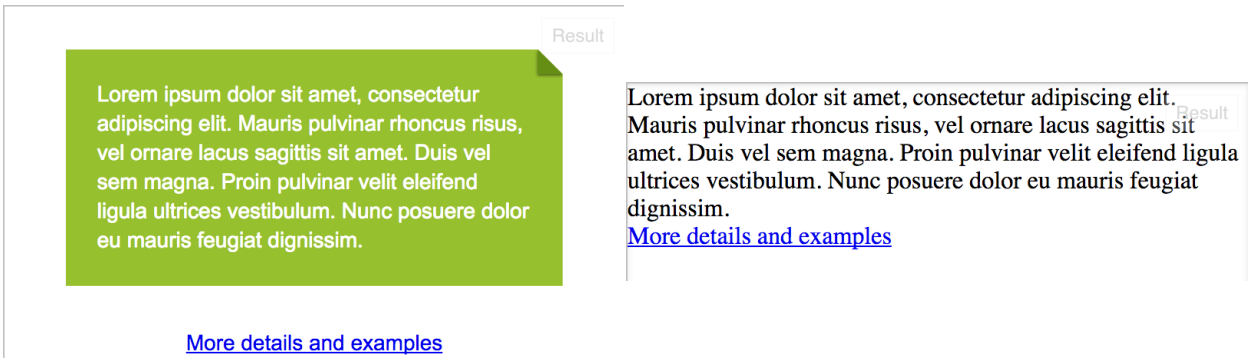


Figure 10: HTML with *CSS* (left) and without *CSS* (right)

In terms of organisation, it is possible to include the code from all three language in one base *HTML* file, but this is extremely undesirable as it lacks modularity and will be hard to test and debug. Instead, it is convenient to keep each type of language to separate files. This makes everything more modular, easier to edit, and individual components will be easier to update or swap out. Methods on dealing with more complex bodies of code will be discussed in a later section.

2.5.3 Web Browser Compatibility

To make things more challenging, various programming aspects such of *HTML*,]*CSS*, *JavaScript* and *DOM* functionality are supported to varying degrees in the different web browsers and versions. This stems from the fact that web browsers are created and maintained by different companies (with differing business goals and implementation strategies). In the late 1990's this proved disastrous as web browsers with different features existed without any coordination between them. However with the creation of the World Wide Web Consortium, matters took a turn for the better. The organisation is lead by Tim Berners-Lee, the inventor of the World Wide Web, and sets out the standards used for web browser scripting, such as *CSS*, *HTML*, *JavaScript* and many more. This happened in several stages and revisions over the years, and is now probably more unified than ever before.

However there still remain small differences, which is only natural when different companies are maintaining these different browsers. For example, *WebGL* has been supported by Google *Chrome* since February 2011, whereas it was only introduced to the *Internet Explorer* with Version 11, which came out in October 2013. Also, not all *CSS* features are universally supported or at least support is lagging behind between browsers. For example when searching for the *CSS* property "box-shadow" www.w3schools.com will list which browsers and versions support this attribute. Luckily the most commonly used elements tend to be supported across all browsers. The http://www.quirksmode.org website aims to provide a comprehensive breakdown of provided features across different browsers. Another issue is that the web browsers' *JavaScript* interpreters can differ in their implementation.

It is anticipated that all these variations will become a source of problems when developing this project and some time will have to be dedicated to test and if necessary reimplement certain sections of the code which do not run successfully in all browsers. A method called 'browser sniffing' could be employed, where the type and version of web browser is determined and the code configured accordingly. However W3C does not recommend this, as there would be too many variations to cater for. Instead they advocate using feature detection, which allows the developer to query for support of a feature such as *WebGL* instead of having to check the browser version when *WebGL* was first supported. This is more elegant and future-proof. Looking at this from a code perspective, detecting a browser and version is done by the *navigator.userAgent JavaScript* command. It returns a string like:

```
Opera/9.80 (Macintosh; Intel Mac OS X 10.6.8; U; Edition Next; en) Presto/2.10.289
Version/12.00
```

This string can be parsed (returning the browser as *Opera*, OS as *OSX* and Version as 12.00),

but will obviously differ widely for each different browser. It is also error prone, depending on the string parsing method employed and assumptions made about the format of the string. Instead, detecting a feature would be done like so:

```
if(!!document.createElement('video').canPlayType === true) {
  // run some code that relies on \textit{HTML}5 video
} else {
  // do something else
}
```

Testing for features like this seems a better approach since the code directly testing for a specific feature that should be called instead of relying on intricate knowledge of which version of which web browser has what capabilites. Web browsers for mobile devices add another layer of complexity to this, but will be ignored for the scope of this project.

2.5.4 Web Frameworks for Single Page Apps

Part of the aim of this project is to create a singe-page application (SPA), which loads into the browser on the client side and does not require complete page refreshes (Osmani, 2013) [5].

As goes for any body of complex code, an orderly and modular structure should be implemented and maintained. The estimated length of code for this project lies in the tens of thousands of lines of code and a multitude of classes. Therefore, it is in no way feasible to just include the *JavaScript* in script tags in the actual *HTML* pages. A more advanced solution is needed.

Thankfully, this requirement for advanced code structuring was generally recognised, and a dominant solution to this issue by the name of Model View Controller (MVC) pattern has emerged. MVC separates the concerns in an application into three parts [5]:

- Models represent the domain-specific knowledge and data in an application, such as specific data container classes. Models can notify observers when their state changes.
- Views typically constitute the user interface in an application (such as HTML markup and templates), but not necessarily so. They observe models, but do not directly communicate with them.
- Controllers handle input (clicks or user actions) and update models.

So in an MVC application, user input is handled by controllers, which then update various models. Views listen to the state of models and update the user interface or visual front end when changes are detected. The definition of MVC frameworks is not always followed strictly and some merge the role of controller and views into one. *Backbone*, which will be used for this project does exactly this. For that reason, these more loosely defined implementations are sometime called MV* frameworks. It would in theory be possible to organise the code for this project in a self-made MV* pattern, but libraries such as *Backbone* are well established and reduce a lot of tedious work that would otherwise be required. *AngularJS*, *CanJS* and *EmberJS* are other popular MVC implementations.

At the time of writing, the *Backbone* library seems to be a good choice to manage the complexity and structure of the project. It was written by Jeremy Ashkenas and is designed for developing SPAs, and for keeping various parts of web applications synchronized. It has been used by big companies to create nontrivial applications, such as *Walmart*, *SoundCloud*, and *LinkedIn*. I have also used it in successfully in a previous group project. In accordance with the MV* pattern, *Backbone* provides Events, Models, Collections and Views. Models represent the information in class-like structures, Collections are lists of Models and Views handle the visual representation on the actual *HTML* site. Views actually generate the *HTML* that goes into the DOM and can be seen on screen. The *HTML* can be generated as strings in the View or through dedicated *HTML* template files, that a View can access. Finally Events are used to keep track of state changes

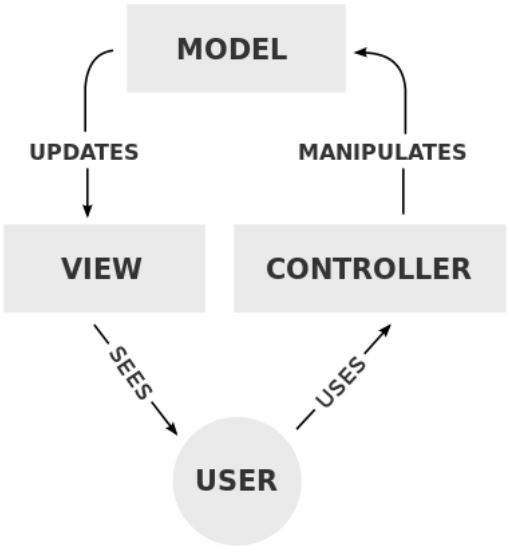


Figure 11: A typical interaction of MVC components (Image taken from Wikipedia [27])

across the website elements, models and views. *Backbone* also provides inheritance for its models, collections and views.

Backbone uses *jQuery* and *Underscore.js*, both of which are libraries that simplify dealing with *HTML* elements. *jQuery* is widely used and allows a developer to conveniently query the DOM for elements such as id or class tags. For example if there is an*HTML* button element with a specified id tag, *jQuery* lets the developer load this element as a *JavaScript* object and work with it in his code. These kinds of *jQuery* objects provide various convenience-based methods for setting its attributes. *Underscore* is another collection of convenient *JavaScript* methods. For example it provides basic methods for dealing with arrays, such as *contains(array, value)*, returning a bool if a value is contained in a list, and *indexOf(array, value)*, returning the index of a given value in an array. *Underscore* is described on its website as

> "(Underscore is) the answer to the question: "If I sit down in front of a blank HTML page, and want to start being productive immediately, what do I need?" ... and the tie to go along with jQuery's tux and Backbone's suspenders." (Ashkenas, [14])

2.6 Graphics for Web Applications

In order to create an interactive browser-based application which is useful for displaying medical image data, the available tools for creating 2D and 3D graphics in a web browser have to be considered. Generally, in the past web browsers have provided several different methods to display 2D and 3D graphics on screen, and only recently with the introduction of *HTML5* and *WebGL* has a widely-conformed standard emerged.

HTML5 is the fifth revision of the *HTML* standard. *HTML5* introduced a number of important features that were designed to facilitate handling multimedia and graphical content on the web without the use of proprietary plugins and APIs. One of these new features is the *Canvas* element, a scriptable graphical display element which is a low level, procedural model that updates a bitmap element. It was originally introduced by Apple [8] . This components forms the basis for most complex rendering of 2D and 3D graphics in modern browser applications.

2.7 2D Graphics

Displaying 2D graphics has long been an integral part of web browsers. Various aspects and methods allow for the generation and manipulation of 2D graphics in a web browser. Aside from simply

displaying an image file, there are various other aspects and methods allow for the generation and manipulation of 2D graphics in a web browser

CSS, as discussed, is used to alter an element's style property, but this is not really suitable for creating intricate 2D graphics, as it is typically used to style the look of *HTML* elements. There are no custom draw commands.

Scalable Vector Graphics (*SVG*) is like *HTML* for graphics [9]. It is a markup language for describing all aspects of an image or Web application, from the geometry of shapes, to the styling of text and shapes, to animation, to multimedia presentations including video and audio. It is fully interactive, and includes a scriptable *DOM* as well as animation. It supports a wide range of visual features such as gradients, opacity, filters, clipping, and masking. The use of *SVG* allows fully scalable, smooth, reusable graphics for a wide range of demands such as games, scientific visualisation and more. *SVG* is natively supported by most modern browsers, and is widely available on mobile devices and set-top boxes. All major vector graphics drawing tools import and export *SVG*, and they can also be generated through client-side or server-side scripting languages.

Finally, the more recent *HTML5 Canvas* is a scriptable graphics element that can be controlled with *JavaScript*, and appears to be the most popular choice of display 2D Graphics on the Web. It has a drawable region which can be drawn to by inbuilt drawing functions. It is a low level model with no concept of a scene graph or *DOM* and can therefore perform very quickly. Dedicated programmers can develop games or even full-featured applications using the *Canvas* API, alone or integrated into *HTML* or *SVG*. It is supported in modern web browsers and even on smart phone mobile devices.

Before the *Canvas* API became common place, there were different web browser plug-ins which would display more interactive graphics and videos. Adobe's *FlashPlayer* and *JavaApplets* were very common for this purpose.

2.8 3D Graphics

3D Graphics are usually defined by a space in Cartesian coordinates in which reside three-dimensional objects, as well as a camera object through which the scene is viewed with help of a projection matrix. 3D graphics is computationally much more expensive than 2D graphics due to its more complex mathematics. On top of that, rendering a 3D scene can mean that the image should be refreshed 60 times a second, which used to pose a challenge for many web browsers.

2.8.1 Pre-WebGL

Broadly speaking, the history of 3D Graphics for the Web can be divided into the time before and after the standardisation of *WebGL*. Before *WebGL* the standard way to display 3D graphics in a web browser were tied to browser plug-ins that the user would have to download locally to their computer. Adobe's Flashplayer used to be one of the dominant plug-ins. Demonstrating one of the issues with the plug-ins is that they would have to be implemented specifically for each operating system. Furthermore, Apple actually refused to support the format, instead betting on the *HTML5* standard. This shows that there are issues with using browser plug-ins, and a more generalised solution was sought for.

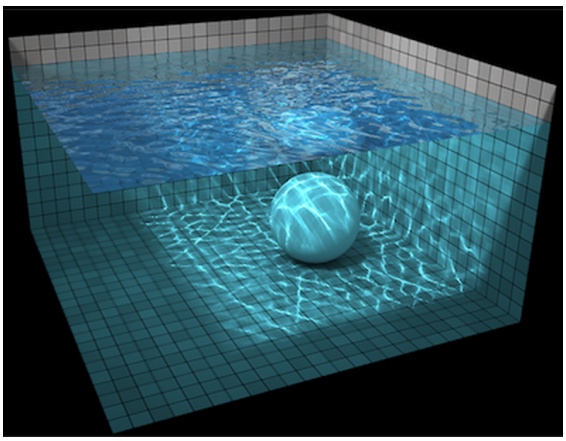
Various other brower-plugins have been created in the quest for 3D Graphics on the web, such as Adobe's *FlashViewer* and *O3D*. *Java Applets* are another alternative, which run on the local machine through the *Java* virtual machine. Microsoft implemented its own plugin *Silverlight*.

2.8.2 WebGL

In parallel with the standardisation of *HTML5*, *WebGL* became more widely used and supported, and helped to reduce the multitude and clutter of browser-plugins. Generally speaking, *WebGL* is a 3D graphics API for the Web. It has been used among others on *Google Maps*, *Autodesk*'s *Autocad Cloud* applications and *Epic Games*' *WebGL* Demo *Citadel*.

The website http://www.awwwards.com [30] hosts a number of impressive showcases of WebGL's capabilities, such as "WebGL Water" by Evan Wallace [28] which features a swimming

(a) Epic Games' Citadel WebGL Demo



(b) WebGL Water

pool and a ball with fluid dynamics and high-end visual phenomena such as reflection, refraction, caustics, and ambient occlusion.

WebGL is exclusively controlled by *JavaScript*. *WebGL* is based on *OpenGL* ES 2.0, the Embedded Systems version of the widely used *OpenGL* standard for generating 3D graphics. The Embedded Systems off-shoot is intended for smaller computing devices such as smart phones and tablets, (e.g. IPhone, IPad and Android phones). Due to its smaller footprint, it was felt that *WebGL* would make it a more consistent, cross-platform and cross-browser 3D API (Parisi, 2014) [2].

WebGL uses the*HTML5 Canvas* element to render into, and uses a specific *WebGL*- compliant context (as opposed to the 2D context used for 2D drawing). It differs in 2D Drawing, that all rendering is done with primitives such as Triangles, Points or Lines. Dedicated arrays called buffers are used to store the data for these primitives, such as vertices and normals. Next, a ModelView matrix needs to be created, which sets the transformation of a primitive or object in the 3D World. Also, a projection matrix is needed which converts 3D points into 2D coordinates of the drawing screen. Finally, a shader needs to be defined which will be attached to the primitives. Shaders are written in their own language called *GLSL*, and are themselves typically comprised of vertex shaders and fragments shaders. Vertex shaders combine the matrices and position of primitives into the final position on screen whereas the fragment shader is used to assign attributes such as color, texture and lighting. Thus, the workflow to render something with *WebGL* onto a webpage is as follows [2]:

1. Create a Canvas element.
2. Obtain a drawing context for the Canvas.
3. Initialize the viewport, synonymous with setting the rectangular bounds of where to draw.
4. Create one or more matrices to define the transformation from vertex buffers to screen space.
5. Create one or more shaders to implement the drawing algorithm.
6. Initialize the shaders with parameters.
7. Draw.

As can be guessed from the steps above, *WebGL* is a very low level API in that it does not provide any high level scene descriptions of scene graphs. All this has to be implemented by developers manually. A large part of its draw and power stem from this fact, but also makes it very time intensive to work with from the ground up. For this purpose, some solutions already exist that build on this to provide a more convenient interface into the 3D capabilities such as ThreeJS. For this project, the *XTK* library plays the same role in that the low level features of *WebGL* can be accessed by a high-level API which can load a volume file as 3-plane slice display in a 3D View.

Nowadays, most browser-based 3D graphics libraries make use of *WebGL* and even libraries that used to require a plug-in have now been ported to use *WebGL*, such as *O3D*.

2.9 XTK Library

As suggested in the project specification, the goal of this project is to use the *XTK* library (or *X Toolkit*), a *JavaScript*-based framework for visualizing and interacting with medical imaging data using *WebGL*. Based on searching for comparable APIs, it is the most advanced JavaScript-based API for medical image viewing that currently exists. It was developed and maintained by Haehn et al of the Fetal-Neonatal Neuroimaging and Developmental Science Center at Childrens Hospital Boston, Harvard Medical School, US ([7]). *XTK* provides an API to load medical images, using *WebGL* for displaying 3D graphics and the *HTML5 Canvas* element to display 2D components. It hides a lot of the low-level elements of *WebGL* and is designed to load and configure medical image data files of various types (including *NIfTI*). The library is well documented and comes with several tutorials as well as a number of example applications, which will be discussed in the next section. It appears to follow closely the suggestions laid out in section 2.2 (Guidelines for Medical Imaging) in that it provides means to display 2D and 3D slice visualisations (Surface Shaded Display and Volume Rendering are supported), with simple mouse-based controls. It allows for loading of label (segmentation) maps which can overlaid on top of the image data files. It also features color tables (LUTs) to display the data. More advanced features such as reslicing a volume along arbitrary axis and rendering particles are also supported.

The *XTK* library also provides a module for creating User Interface (UI), that can be overlaid on top of the image viewers, with easy controls to connect the image content with the UI controls.

2.9.1 Use of Google Closure Library

The *XTK* library is written in *JavaScript*, but uses the *Google Closure* library to compile the code into minified code. This serves to reduce variable and function names and generally is designed to make the code run faster. The drawback is that it become harder to decode, since variable and function names are changed to be meaningless to the human eye. For functions and attributes that are meant to be used from outside the library, custom setters and getters have to be written. *XTK* also uses *Goog matrices* for their graphics computations. Additionally the *WebGL* portion of *XTK* makes use of *GLSL* for its shaders, but commands are wrapped as strings in a *JavaScript* file.

2.9.2 XTK Classes

The library is comprised of a host of different classes. Since it would take too long to explain every class, four key concepts will be discussed (*X.renderers*, *X.volumes*, *X.slices* and *X.labelmaps*) to give an insight into *XTK*'s workings. The *XTK* website has a complete if somewhat out of date list of its classes, attributes and methods, if the reader wants to learn more about it. Figure 49 shows a big picture UML class diagram of how the various classes interact with each other.

An *X.renderer* is an abstract virtual class and both *X.renderer3D* and *X.renderer2D* inherit from it. The renderer has a private attribute called *topLevelObjects* which is an array of *X.objects* that have no parents themselves. This matters as inherited *X.objects* such as *X.volume* have nested *X.objects* as member attributes. The *X.renderer* class also has a private objects class, which is an array of all *X.objects* that are linked to this renderer. Next, an *X.loader* object is also assigned to the private *loader* attribute. This loader will be used for any loading of file object, as will be discussed in more detail later in this subsection. The *X.renderer* class also has an *X.camera* class. This is used to view the renderer content, and viewing transformations such as panning, zooming and rotation are handled by this class. There exists an inherited 2D and 3D camera class. Then there is an *X.interactor* assigned as well. This class handles all mouse and keyboard inputs.

The inherited *X.renderer2D* class has extra attributes to specify which slice is being rendered, such as *currentSlice*, *orientation*, *sliceHeight* and *sliceWidth*. It has a *framebuffer* array attribute which stores the current slice texture information which will be rendered to screen. It also has a *labelFrameBuffer* which is a secondary buffer that is used to render a labelmap if one has been loaded. This is in essence an A over B compositing effect. The *X.renderer3D* class has different internal attributes since it does its drawing through *WebGL* technology. It contains various number attributes of the dimensions of the loaded objects, as well as an *X.shaders* object and various buffers

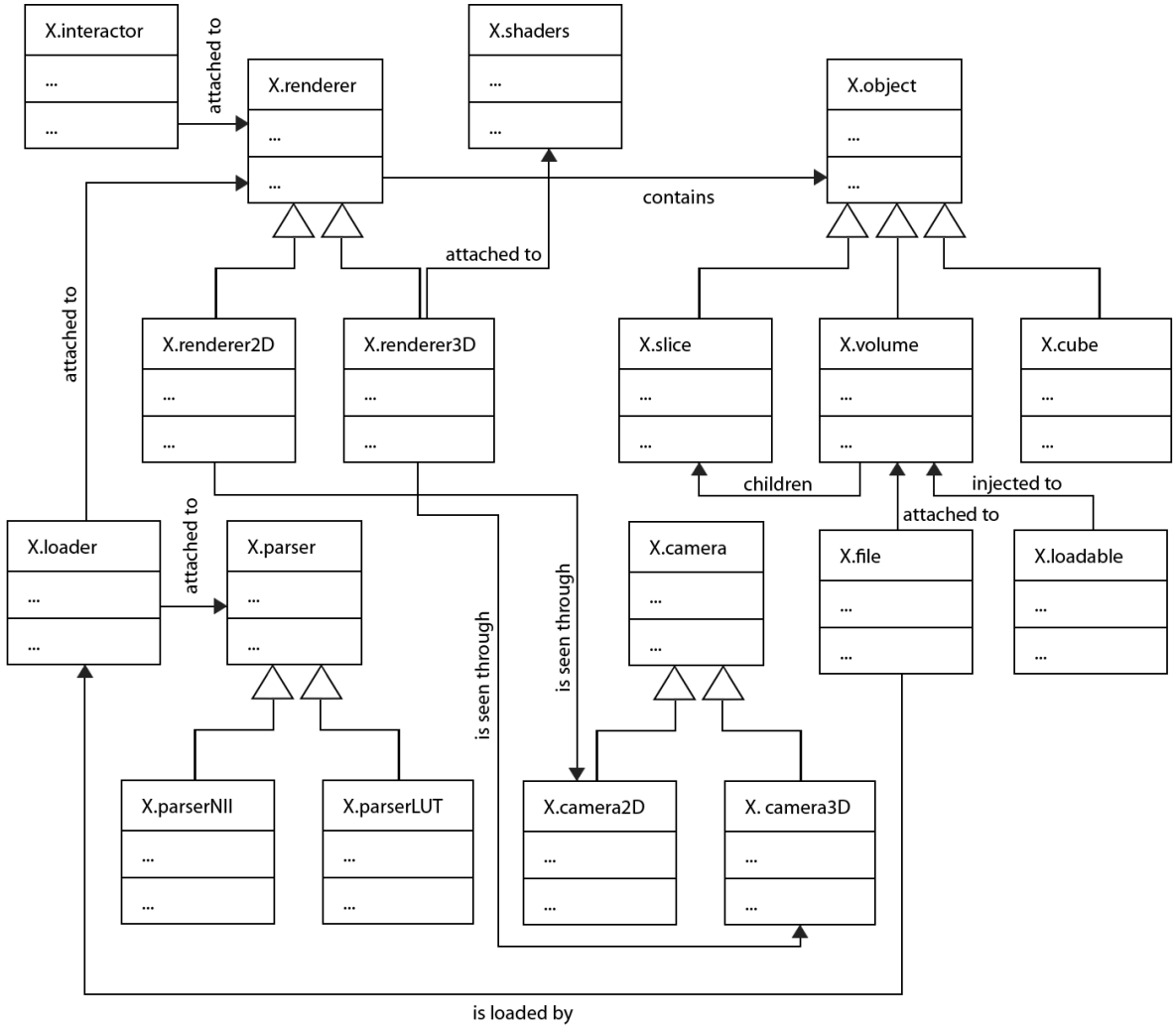


Figure 13: Flow of Control of *XTK* renderer

of the form *goog.structs.Map* to store vertex, normal, color and texture information of the currently loaded objects.

There are various methods in the *X.renderer* classes. *Init()*, *render()* and *update()* warrant a closer look. The *init()* function is called in the beginning to set up main aspects such as the *X.interactor*, *X.camera*, *X.loader*, *HTML Canvas* and *Context* on which to draw on. The *render()* function deals with the actual drawing of content onto the *Canvas*. The *X.renderer3D* and *X.renderer2D* classes naturally differ in their implementations of this function, but both call on the parent's *render()* function to continuously redraw the frame. *X.renderer2D.render()* function reevaluates if the current buffer needs to be recalculated and if so does it inside the scope of the class, whereas the *X.renderer3D.render()* function pushes any changed data to the GPU for drawing.

Generally, *XTK* uses the concept of injections to add extra features into various *X.classes*. A selection can be see in Figure **??**. *X.loadable* supplies attributes to store file data. *X.displayable* offers several attributes useful for displaying in *Canvas*, such as *_color*, *_points* and *_texture*. Finally, *X.thresholdable* enables the setting of different threshold values, as is used in the *X.volume*.

X.slice is a type of *X.object* and inherits a host of attributes and functions from it (see Diagram **??** for UML diagram). It stores attributes for a current slice, such as orientation vectors (*_front*, *_up*, *_right)*, *_center*, *_width* and *_height*. Through the *displayable* injection of *X.object*, it also has various useful attributes for being displayed in the *HTML Canvas*, such as color, points, opacity and texture. The *texture* attribute actually points to an *X.texture* which is where the actual pixel

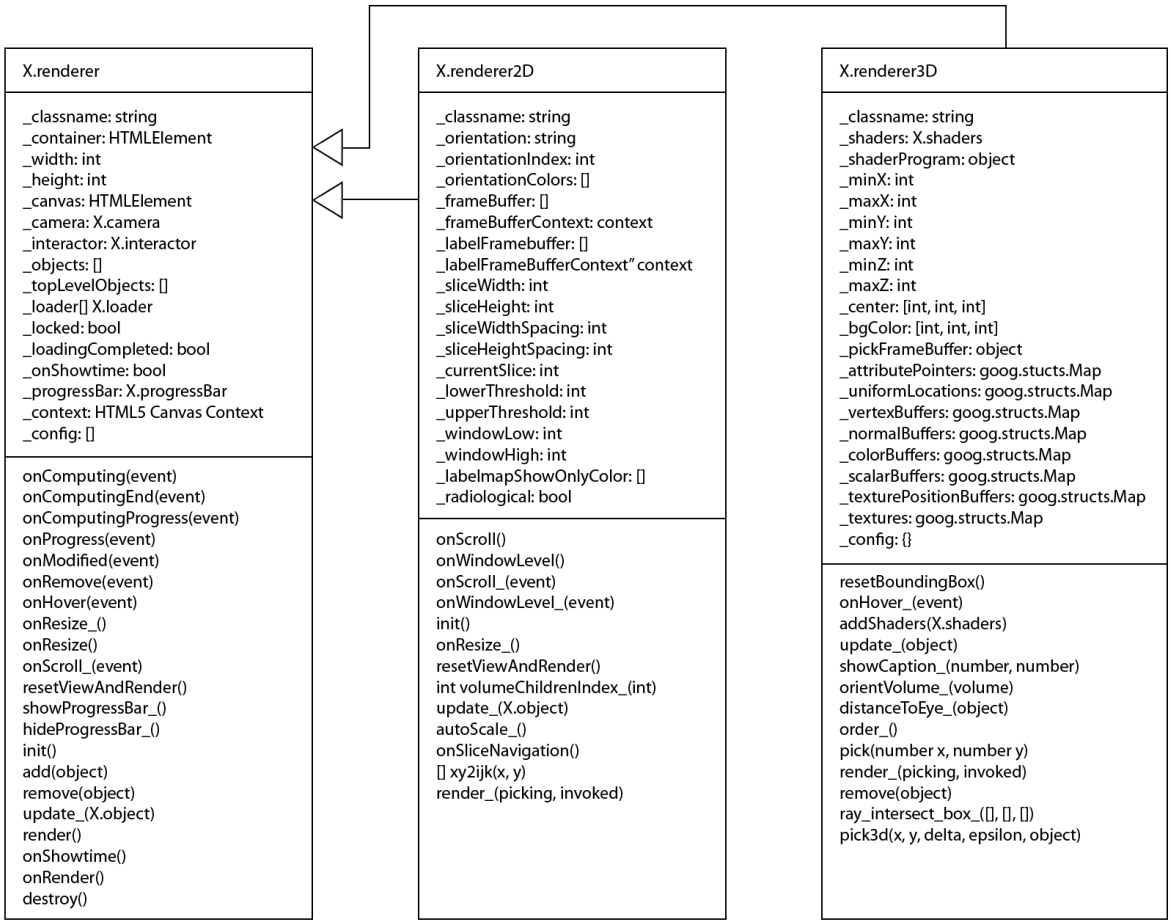


Figure 14: Full Class Diagrams for renderer.js, renderer2D.js and renderer3D.js

data is being stored for the current *X.slice*. It is this texture data that will ultimately be displayed in the *X.renderer*.

The *X.volume* is a more complex type of *X.object*. Therefore it can be added to a *X.renderer* (the *add()* functions enters it into the *_topLevelObjects* array) and inherits all the attributes and methods from the *X.object* class. Volumes are also injected with the *loadable.js* code and *thresholdable.js* code. An *X.volume* has a variety of attributes that deal with the currently loaded data, such as current indices (*_indexX*, *_indexY* and *_indexZ*) and *range* which specifies the maximum indices. It has *_windowLow*, *_windowHigh*, *_thresholdLow* and *_thresholdHigh* (the last two from *thesholdable.js*) which dictate the window and threshold settings. As for all attributes that are to be accessed from outside the API, setter methods have been written which take care of internal logic. It is the setter methods for *window* and *threshold* that are being called through the Levels Panel when adjusting the sliders. The *X.volume* has the inherited array attribute *children* (originally from *X.object*) in which it stores all the precomputed *X.slices*. A precomputed *X.slice* is generated when the slice index is set to an index that has not been visited yet since instantiation. In this way, the *X.volume* will buffer more and more slices throughout scrolling. Once the *X.slices* have been buffered, lookup is noticeably quicker. When the *X.volume* is set to an uncomputed sliceIndex, different functions are called than when a precomputed slice is present, including forcing a redraw in the *X.renderer* class.

The *X.volume* also has an attribute to store an *X.labelmap*. These are actually inherited from *X.volume* (with very few overloaded functions) and function mostly the same way. One of the key differences is that at render time for *X.renderer2D*'s, the *X.labelmap*'s slice texture is rendered to a different buffer than the *X.volume*'s slice texture. The buffers are combined just before the pixels are rendered on screen. By default, the *X.slice*'s textures for an *X.volume* get clamped to a single

colour channel (presumably because medical images tend to be monochrome) whereas *X.labelmaps* get the full RGB range, since coloring is essential to the purpose of labelmaps.

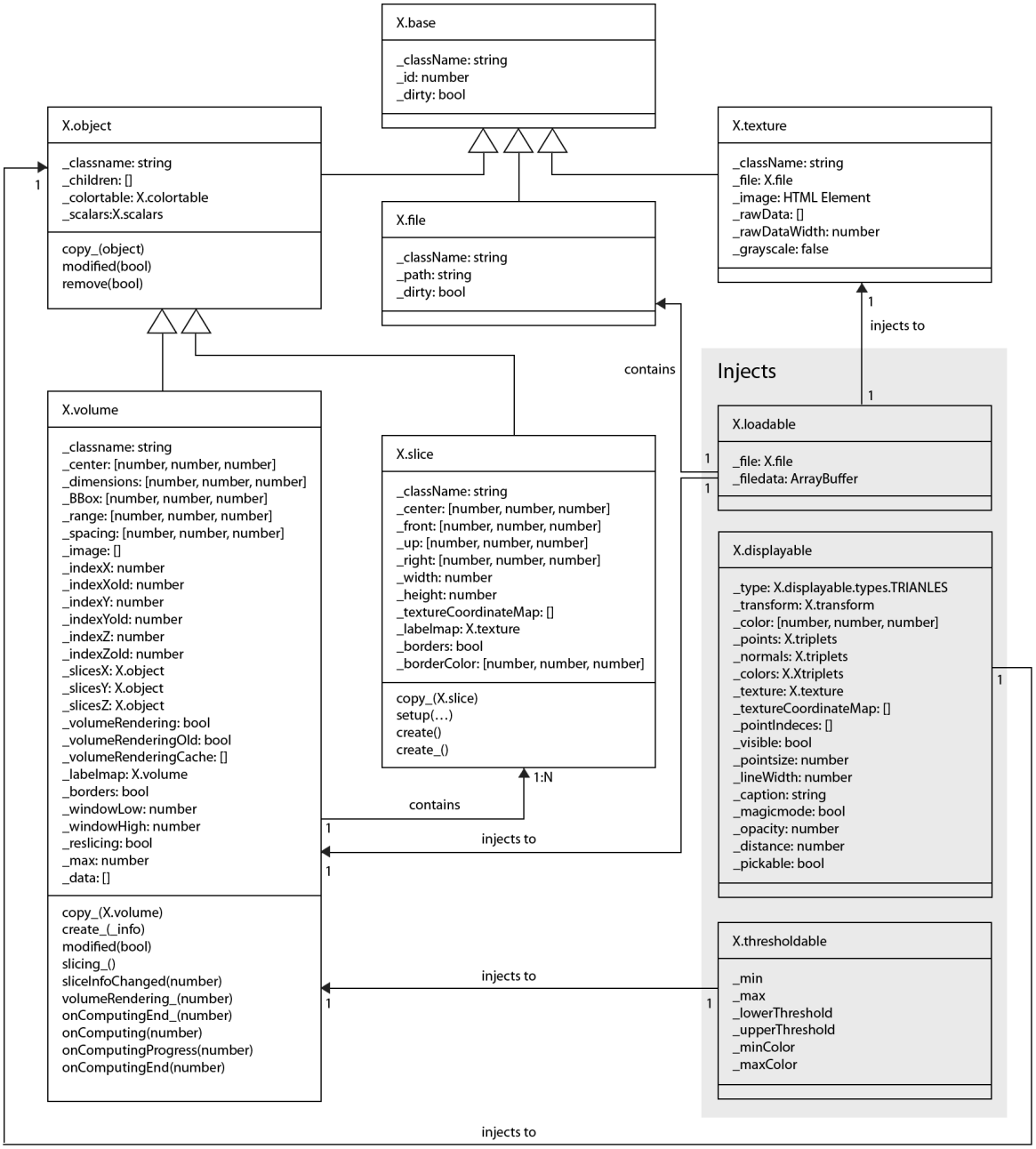


Figure 15: UML Class diagram for X.volume and dependencies

More details on how the *XTK* library works will be discussed in the Implementation section. For now, it should be clear that the *XTK* library provides a number of complex classes that support a lot of the desired functionality that one would want in a medical image viewer, and that it is a very promising starting point for this project.

2.10 Web Applications for Medical Image Viewing

As for the desktop, there also exist some Medical Image Viewing applications on the Web. A few of these have been tested and analysed for features and ease of use.

2.10.1 SliceDrop

The *XTK* library web pages provide links to example applications. One of these is *SliceDrop* [**?**] which was written by the *XTK* developers and appears to be a testing bed for many of *XTK*'s features as it is often referred to on the *XTK* section of the *Stackoverflow* forums. It provides part of the desired functionality and shows off the possibilities provided by *XTK*, but also outlines some of its short comings. Like the desktop software *MITK*, the user can load a file and view it in the standard 4 views, split into 3D and 2D windows. The layout of these views is somewhat customisable, although not to the extent of the desktop software *MITK*. Once a file has been loaded, it can not be replaced with another one. The user can interact with the images by scrolling the mouse wheel, which changes the index of the current slice. An interesting feature which has been added lately is the incorporation of popular file sharing service *DropBox* to allow users to share files more easily across the internet.

There appear to be some bugs in *SliceDrop*. When loading *NRRD* files, as the image is offset and not centered. The image brightness controls seem not calibrated correctly, as with little mouse interaction, the brightness will clamp to white or black and can not be reset other than restarting the program (refreshing the page for web-apps). Also the viewers provide some functionality which is not clearly communicated to the user, such as holding the 'shift' button and move the mouse will adjust the slice index of the other viewers. In general *SliceDrop* has less than the minimum functionality required for this project, but shows the potential of the *XTK* library.

2.10.2 Brainbook

Brainbook [18] is a web-app which builds on and extends the features of *SliceDrop* by adding painting tools. It allows the user to paint onto a given file with a standard brush a number of auto selection tools which will fill out a region in 2D or 3D space. However paining tools seem inexact and behave in a fashion that can be hard to understand for a user. The web app offers to save the file, but at time of writing this feature was not working correctly. Since heavily based on *SliceDrop*, it features similar advantages and disadvantages, but provides the painting functionality that is desirable for this project. Therefore it should be of benefit to study this implementation.

2.10.3 Papaya

Papaya [19] is a *JavaScript*-based medical image viewer with limited functionality. It has a clean, simplistic, uncluttered design, is easy to use and gives good visual feedback to the user. It successfully adheres to the user control guidelines outlined in section 2.1. There are customisation options for label maps, windowing and rudimentary compositing. It also offers a number of preset colortables. However it only displays 2D views of slices and does not allow removing of files. Zooming and scrolling through the slice indices are both mapped to the middle mouse scroll key, and has to be set explicitly in the preferences which seems cumbersome. It also uses the *HTML5 Canvas* element to render the 2D data.

2.10.4 Other

The website `http://www.idoimaging.com` lists a number of other available browser- based viewers.

2.10.5 Summary

In general, it appears that browser-based medical image viewers lack far behind their desktop counter parts in terms of feature richness and ease of use. *SliceDrop* seems like the most impressive of the applications that were tested, as it successfully displays 3D (including volumetric renderings) and 2D views of the medical image data. This will act as the primary comparison for this project, and hopefully it will be surpassed with more features, interactivity and customisation options. *Brainbook*'s segmentation drawing feature hints at the potential of adding this feature with the *XTK* library, but is ultimately let down by the uneven implementation. *Papaya* is impressive in

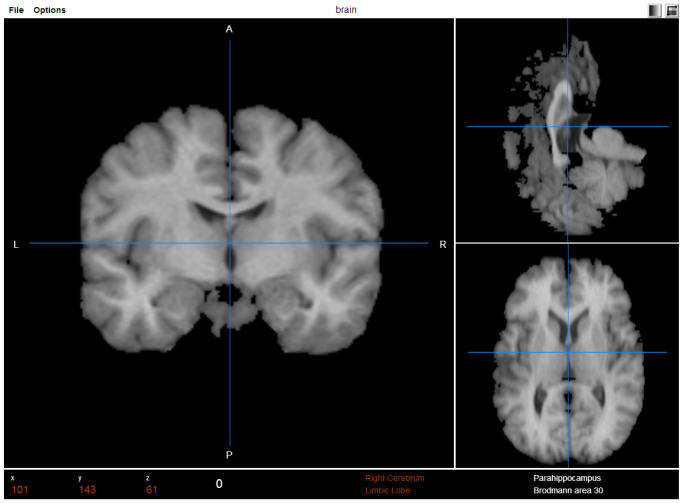


Figure 16: Papaya User Interface

its clean design, polish and UI, even if it does not feature a 3D View. A goal of this project will be to unify and improve the features of these web-apps into one compelling product.

3 Implementation

This section will cover the implementation of *ScanView*. In order to give sufficient context for the discussion of implementing individual features, the first section will cover the full list of final implemented features of *ScanView*. Following that, a visual breakdown of the Layout and UI elements will be presented, which again should give the reader a more complete picture when discussing the implementation of features. Then the general development strategy will be discussed as well as the general structure of the code of the project. A detailed section explaining the implementation of individual features will follow. Finally, I will discuss the contributions of the various external libraries used on this project.

3.1 Implemented Features

At the end of the project, the software has been deployed to this website:

`http://www.davidbasalla3d.com/MscComputerScience/IndividualProject/Code/index.html`

and it has the following features:

- Display Layer Management
  - Creation/Deletion of Display Layers
  - Loading Medical Image Data File per Display Layer
- Loading of NII Volume Files (one per Display Layer), including load error management
- Changing of Brightness
- Changing Image Threshold
- Changing indices of NII Volume File
- Viewing of NII Volume File through four bespoke cameras (X, Y, Z and 3D)
- Custom Layouts of the four different cameras
- Ability to pan/zoom and rotate the NII Volume File through the cameras item Ability to traverse by left-click and drag on a 2D-Renderer will update the other 2D Renderers to the current indices
- Refocus the cameras by pressing 'F'
- Two buffers to hold different Display Layers
- Buffer Opacity and Buffer Swiping to compare the two Display Layers
- Changing the color lookup table for a Display Layer, currently three supported Lookups (None, ID's and Heatmap)
- Toggle for Volumetric Rendering of the Data in 3D Camera View
- Interactive Annotation Management
  - Creation/Deletion of Annotation(s)
  - Support for multiple Annotations
  - Customisation of Label, Color and Position of Annotation
  - Saving out of Annotations as *JSON* file (via file download)
  - Loading and Importing of Annotation *JSON* files

  - Setting visibility of Annotations

- Labelmap Management
  - Loading of Labelmaps
  - Changing opacity of Labelmap
  - Changing color lookup of Labelmap

- 'Cine' Mode for playing through Slices in a Camera View

3.2 Layout and Input Breakdowns

3.2.1 Main Page Layout

The main view that the user is presented with is split into four different components. The Navigation Bar (in blue colour) contains the links to other sites such as Tutorials, Sample Data and the About webpage. It also contains the Layout Selector buttons where the user can chose a layout for the View Panels. The Display Layer Panel allows for managing of Display Layers, which are synonymous with loaded Volume files. Each Display Layer can load a Volume File. The Levels panel contains various controls for affecting the currently selected Display Layer and its inherent Volume File. Finally the Viewer Panel contains the four different Camera View Panels that each Volume is displayed in. They are the perspective 3D View, and the orthogonal X, Y and Z views.

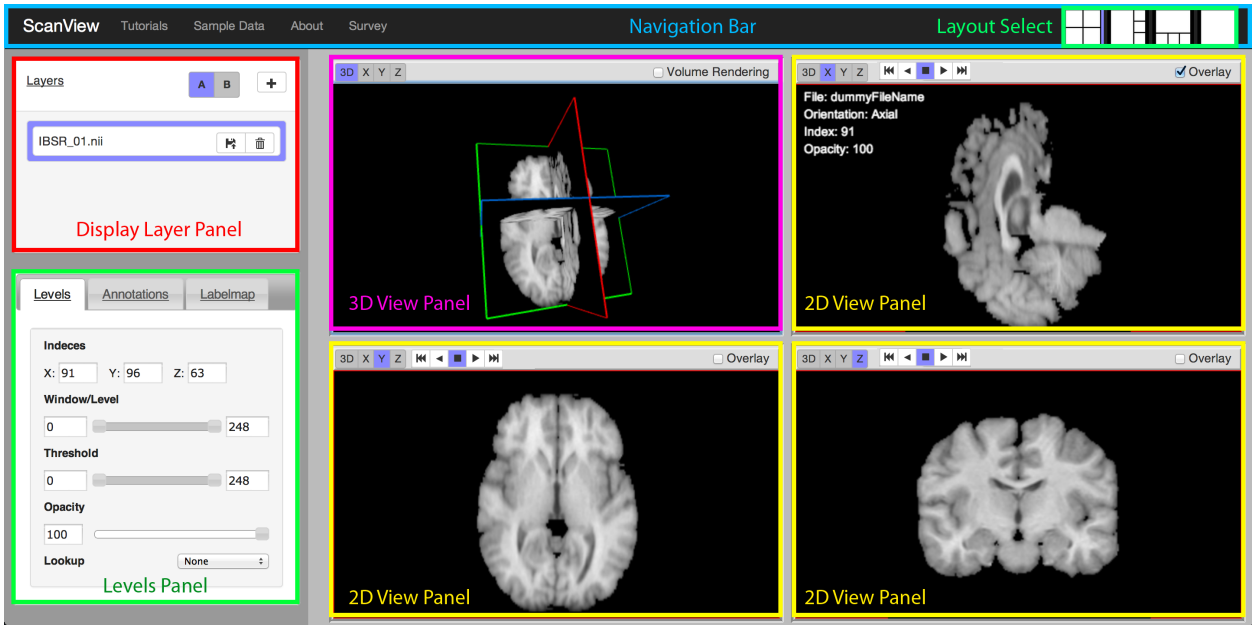


Figure 17: General Panel Breakdown

3.2.2 Camera View Panels

The Camera View Panels are the viewing windows for displaying the visual representation of the loaded Volume File. Internally they copy pixels from invisible *XTK* Render Panels. The View Panels hold a top bar with options to customise the viewer (e.g to specify which camera to look through). The View Panels also contain an *HTML5 Canvas* element which does the drawing of the Volume File. The View Panels come in two types. The 3D View Panel displays the 3D volume with the three orthogonal slices combined as interlocking planes. There is an option to toggle Volume Rendering on, a feature of the *XTK* library. It uses *WebGL* and will only work on computers that support *WebGL*. The 2D View Panels render the individual orthographic (ie. axial, sagittal and coronal) views of the Volume File. They include a set of buttons for playing or reversing through

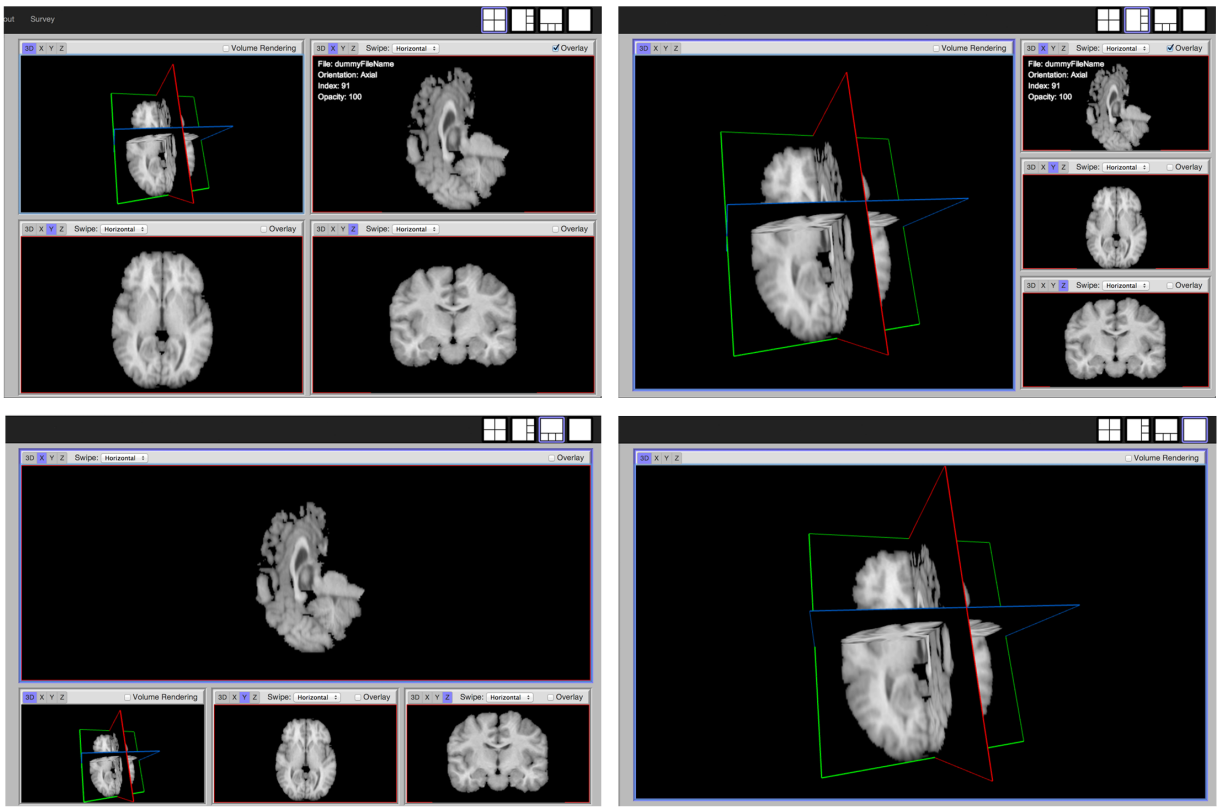
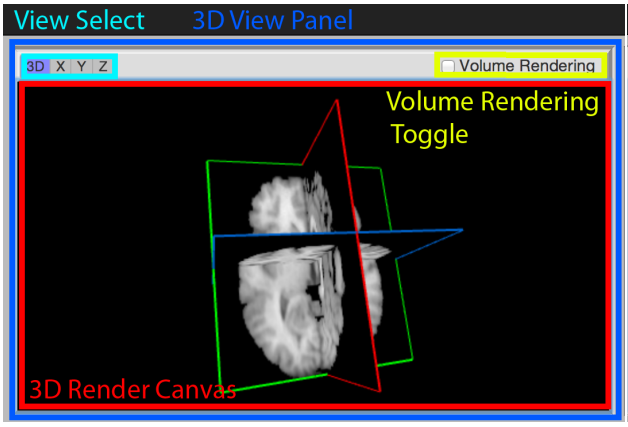


Figure 18: Four Different Layout Options

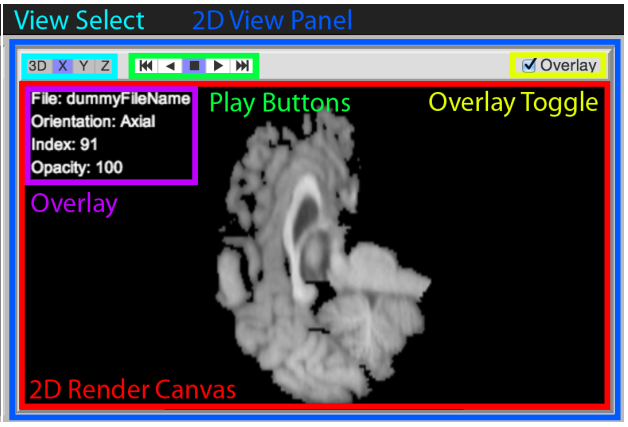
the current slice stack. Furthermore they have a toggle for Overlaying Information into the Viewer about current filename, orientation, index and opacity.

3.2.3 Display Layer Panel

The Display Layer Panel allows the user to manage Display Layers by creating, deleting and modifying them. Each Display Layer can load a Volume File, which will be displayed in the View Panels. Multiple Layers can be created and will highlight blue when selected. Additionally there is a Buffer Toggle Button which allows to switch the viewer between Buffer A and Buffer B. Each Buffer can hold a different Display Layer. This becomes useful when adjusting the Opacity of a Display Layer and it will reveal the Display Layer in the B Buffer. When Buffer A and Buffer B hold the same Display Layer, dialing down the opacity will fade the View Panels to black. Also the Buffer Swipe feature (hold Ctrl and move mouse over a 2D View Panel) plays into this.



(a) 3D View Panel Functionality



(b) 2D View Panel Functionality

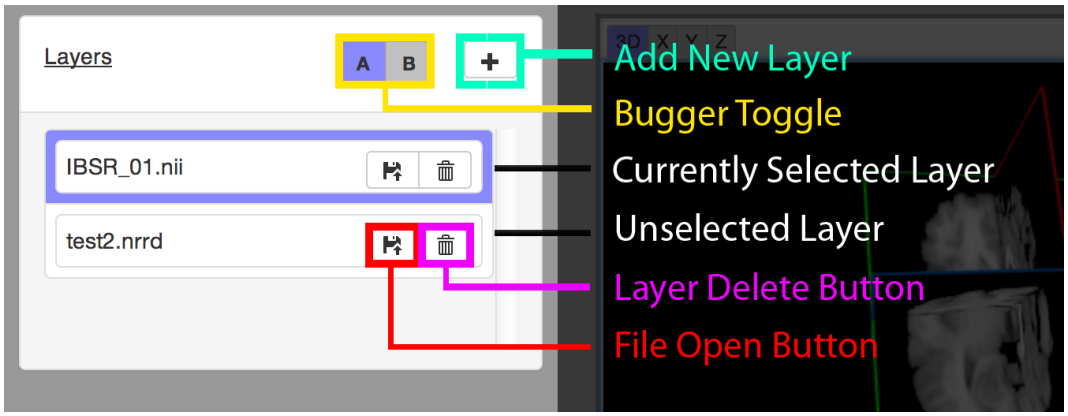
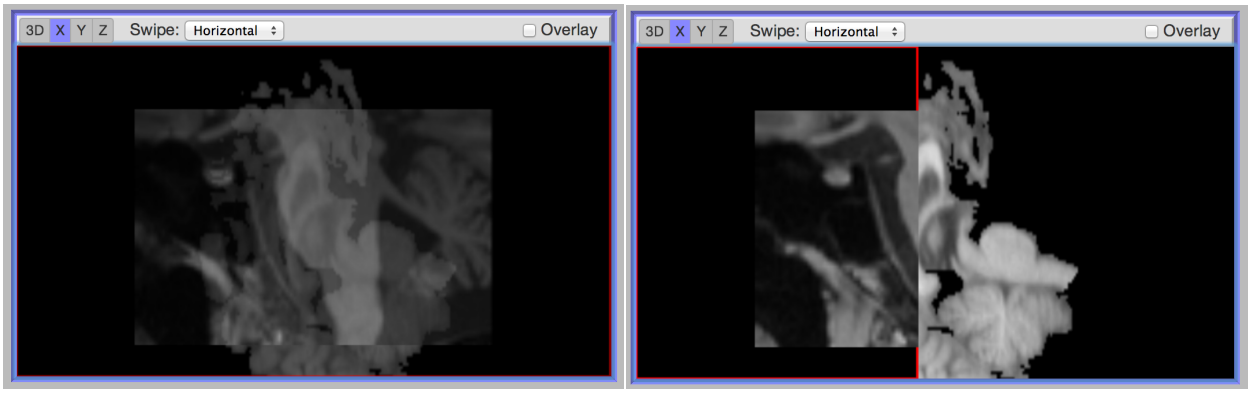


Figure 20: Display Layer Panel Functionality



(a) 50% Opacity on Buffer A with revealed Buffer B (b) Horizontal swiping between two different Buffers

In terms of mouse and keyboard input controls, the 3D Render Canvas supports the common operations of panning, zooming and rotating via mouse controls. Pressing 'F' on the keyboard will reset the camera to the initial setting. Pressing 'v' will toggle the Volume Rendering toggle. Likewise for the 2D Render Canvases, panning and zooming are supported, as well as 'traversing'. When the user clicks the left mouse button and floats the mouse across the slice rendering, the indices will update in the other viewers accordingly as to where the mouse pointer is in the selected View Panel. Also, pressing 'f' will reset the camera. Pressing 'o' will toggle the Overlay toggle.

3.2.4 Levels Panel and Tab

The Levels Panel contains separate tabs that enable further customisation of the currently selected Display Layer. In the Levels tab, a number of convenient controls are provided. Firstly, the Index Controls provide feedback on which Slice Indices are currently set. These can also be altered via the number input.

Next, the Window Controls can be used to control the Brightness of the loaded Volume File. These attributes hook straight into *windowLow* and *windowHigh* attributes of an *X.volume*. Likewise, the Threshold Controls allow for a clipping of a Volume File and is also provided by default by the *XTK* library.

The Opacity Controls allow the user to set the opacity of the currently loaded Display Layer. Depending on what is loaded in the other Buffer, this will either reveal the Volume File from other Display Layer or fade to black.

Finally, the Colortable Controls allow for switching the Colortable to one of three options (None, IDs and Heatmap). This changes the color value lookup at render time, making the Volume File look different with each option. IDs is generally intended for use with Labelmaps, whereas Heatmap can be used for any Volume File.

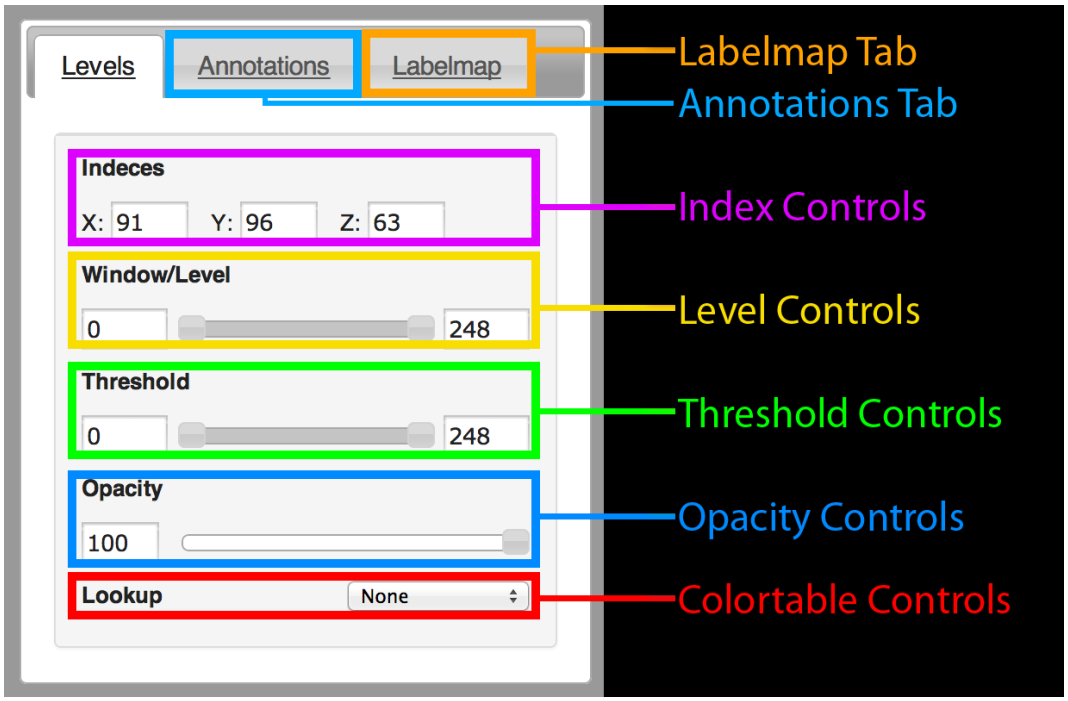


Figure 22: Levels Tab Functionality

3.2.5 Annotations Tab

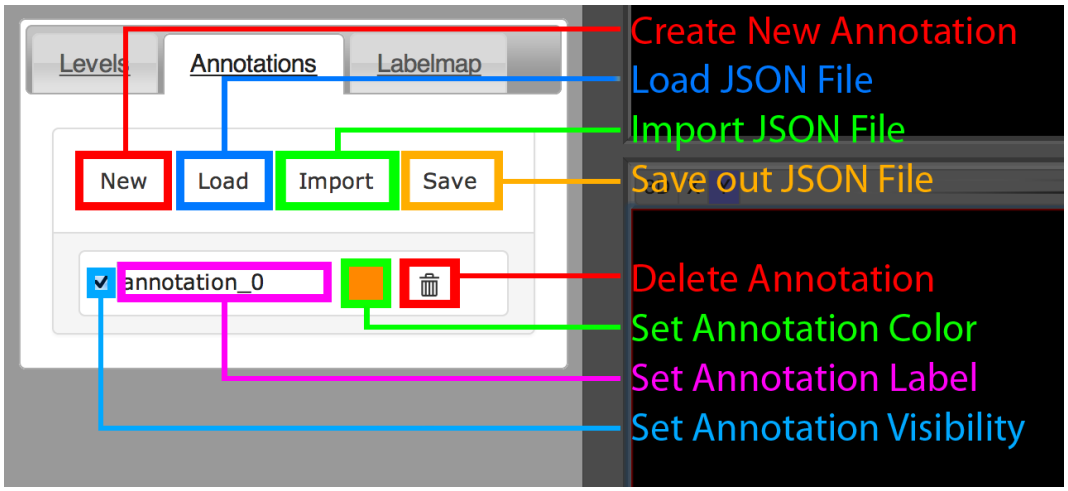


Figure 23: Annotation Tab Functionality

The Annotations Tab allows the user to create, load, customise and delete Annotation Layers. When the user hits 'New', a new Annotation Layer is created, based at the current Volume File indices position with a pre-specified width, length and breadth. The user can then toggle the visibility, change the label and color. Clicking on the color button will open a Colorpicker UI. Finally each Annotation Layer has a Delete Button. Multiple Annotation Layers can be created. The UI allows for loading (via 'Load') a *JSON* file which will remove all current Annotation Layers and create the Layers as specified in the *JSON* file. Clicking 'Import' will add Annotations from a *JSON* file to the current Annotations. Lastly, clicking the 'Save' Button downloads the current Annotations in *JSON* format to the users computer. This would otherwise cause a 'Save As'-style UI modal screen to pop up, but web browsers do not supply this feature for security reasons.

A visible Annotation is displayed as a rectangular volume in the View Panels, colored as per the color specified in the Annotation Layer, with the Annotation label being displayed above it in the 2D Panel Views. The corner points are selectable and manipulatable in the 2D Panel Views.

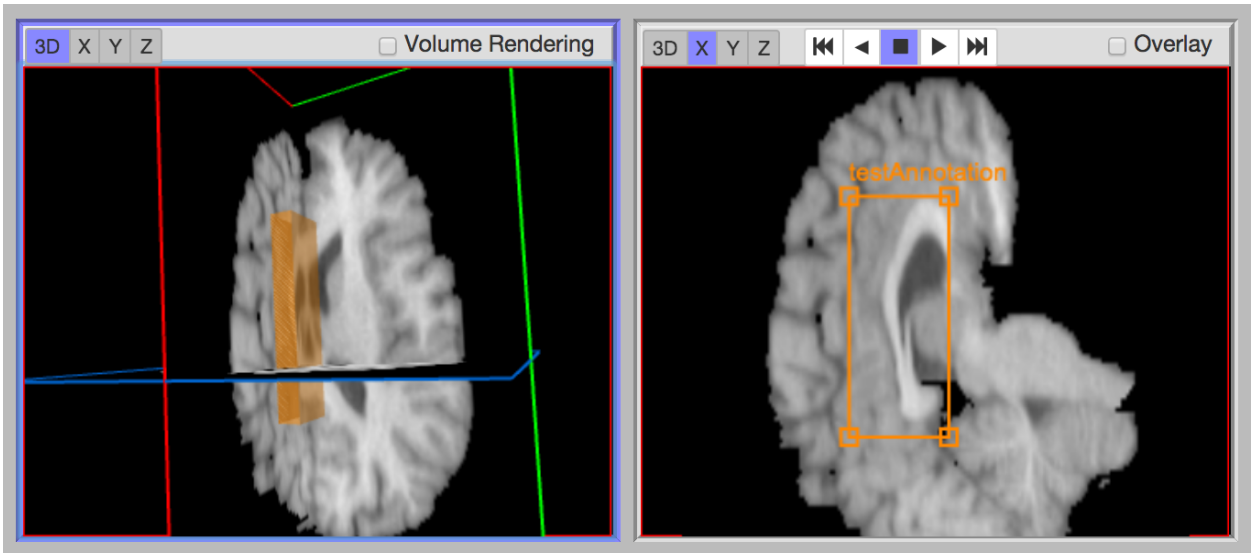


Figure 24: Annotation in 3D (left) and 2D View (right)

Moving one point will move its neighbor points accordingly to keep the rectangular shape. There is a fixed limit to how small an annotation can be made. When a Annotation point is selected, it's according layer is highlighted blue in the Annotation Tab.

3.2.6 Labelmaps

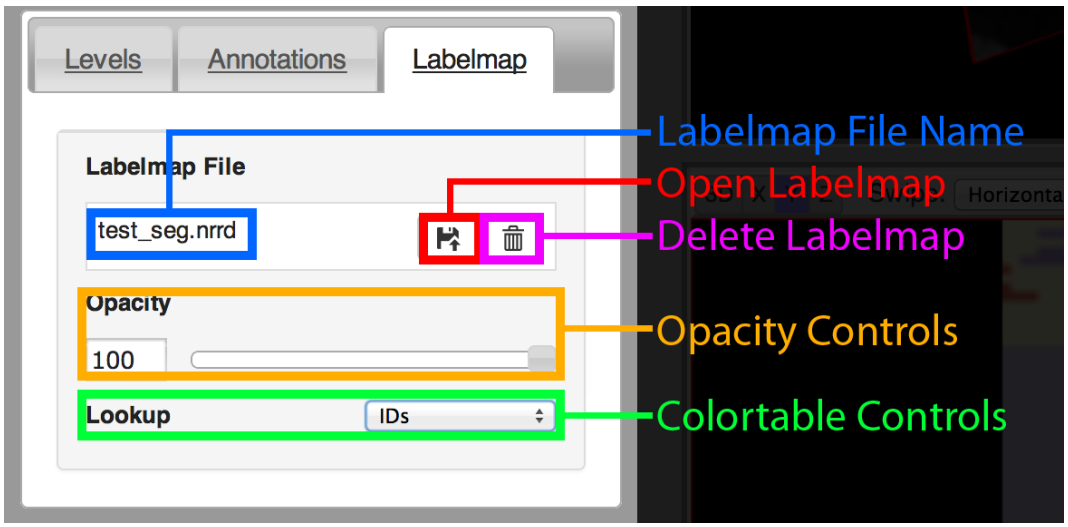


Figure 25: Labelmap Tab Functionality

Labelsmaps can be loaded via the Labelmap tab. The UI allows the user to load one labelmap. There is a Opacity Control for setting the Labelmaps opacity and a Colortable Selector provided. The Colortable Selector provides the same three options as for the Levels tab.

3.3 Development Strategy

Generally the development strategy for this project was to program in modular fashion, tackle problems in a naive fashion first to test for any big problems or in other words to create prototype solutions first. Once a certain prototype milestone had been achieved, some time was spent on refactoring code based on insights learned. A Todo list was maintained throughout the project against which items were ticked off as completed when done. Regular meetings were held with the project supervisor Dr. Ben Glocker to ensure that the project was on the right track. As this was an

individual project and no other programmers were involved, team organisation or communication was not an issue.

3.3.1 Meetings with Supervisor

Meetings with the Project Supervisor Dr. Ben Glocker were held every one to three weeks, based on availability. Written notes were kept as record. Generally we discussed my progress up to date, and which features I should focus on in future. The meetings provided valuable feedback and direction, since Dr. Glocker is an expert in the field of Medical Imaging, has written his own Medical Image Viewers and could provide relevant Software and Data samples. He was treated as the client for this project.

3.3.2 Test Data

My supervisor provided a host of test files for me to use while developing this tool. These varied from smaller files of a brain scan (4.2 Mb) with a labelmap (4.2 Mb), to the largest scan file of an abdomen (86.5 Mb). Additionally I downloaded some sample data from the *XTK* tutorials. This set included the main file (286 Kb), a labelmap (19 Kb) and a colortable (6 Kb). Furthermore I downloaded more samples from the official *Nifti* website. For testing new features, I would test the thorax file set, as well as the brain scans, since these files took less time to load. Periodically I would test the other Nifti files to make sure the code worked with different and larger files. For colortables, I downloaded some from the *3DSlicer* wikipedia site in their default format, and later amended these to work as hardcoded *JavaScript* arrays.

Dr. Glocker kindly also supplied a number of desktop programs for medical image viewing that I could use for inspiration, such as *Imview* and *MIT3K*.

3.3.3 Coding Journal

As the project grew more complex, I started writing a Coding Journal in which I would write down any relevant ideas or thoughts. Since working on one aspect usually took more time than thinking of ideas for other aspects, this proved quite valuable in recording them, organising them and going back over later to make sure I had not missed anything important.

3.3.4 StackOverflow

Whenever I was stuck on a problem, checking the popular *StackOverflow* forum for solutions proved invaluable. At time of writing, there are 15,957 questions tagged with 'Backbone', 521,973 questions tagged with 'JQuery' and 681,696 questions tagged with 'JavaScript', so it provided a vast resource for fixing problems. Even for *XTK* there are 139 Questions tagged, which is relatively small given the other numbers but still useful due to the occasional answer of the *XTK* authors themselves. Three of the questions tagged with 'Xtk' have been asked by me.

3.4 General structure

The initial plan was to build a UI with custom features (the front end) using the *XTK* library (the back end) through the API. The front end would be written in *JavaScript*, *HTML* and *CSS* and would be structured with *Backbone*. The individual *HTML* elements would be stored in separate template files, which the *Backbone* Views would render to the web browser. *Twitter-Bootstrap*, *jQueryUI* and custom-made *CSS* style elements would be used to style the look and feel of the front end.

The hope was that the *XTK* library would provide all necessary functionality required for the front end and so it would not be necessary to alter the library itself. However it soon became apparent that this would not be the case, and modifications had to be made. Still, access to the back end would be handled through a separate dedicated *Backbone* View.

The project would include a number of different libraries which would be dependent on each other. Since some of these libraries exist on remote web servers, the loading has to happen asynchronously. This means that there is no guarantee that a library has loaded when it is needed. For example *Underscore.js* and *jQuery* will need to load before *Backbone.js* is used. Luckily this is a well recognised problem, and a library called *RequireJS* is a great solution to it. It provide the ability to asynchronously load nested dependencies. Traditionally *JavaScript* files or modules are loaded sequentially, where the order matters in case a module depends on another module. *RequireJs* will make it possible to split up everything into neat modules and templates and facilitate testing of all the components.

3.5 Implementing Features/ Implementation History

This subsection will deal with the implementation of general and specific features of *ScanView*. A more detailed explanation of *XTK*'s inner working will follow in the next section, where I discuss my experiences of the various libraries that were used on this project.

3.5.1 Creating the single web app structure

I started the project with one of the tutorials from the *XTK* website. Tutorial 13 had a very basic setup that was similar to what I intended for my project - a four panel rendering of a Volume data file. The tutorial has less than 100 lines of code, with half of it used for specifying an inbuilt GUI. Since I wanted to create a custom GUI and generally have a much bigger codebase, first I need to ensure that I had a solid structure for my project. This is where *Backbone* and *RequireJS* came into play. Since I was not entirely familiar with either of them, it took a few days to get up to speed and try out a few tutorials to familiarise myself with their usage. After a few days I had managed to create a frame work that mimicked the tutorial, but was running in the confines of *Backbone* with proper Models, Views according to the MV* pattern and managed dependencies through *RequireJS*. Figure 26 shows a UML-style representation of the adopted architecture.

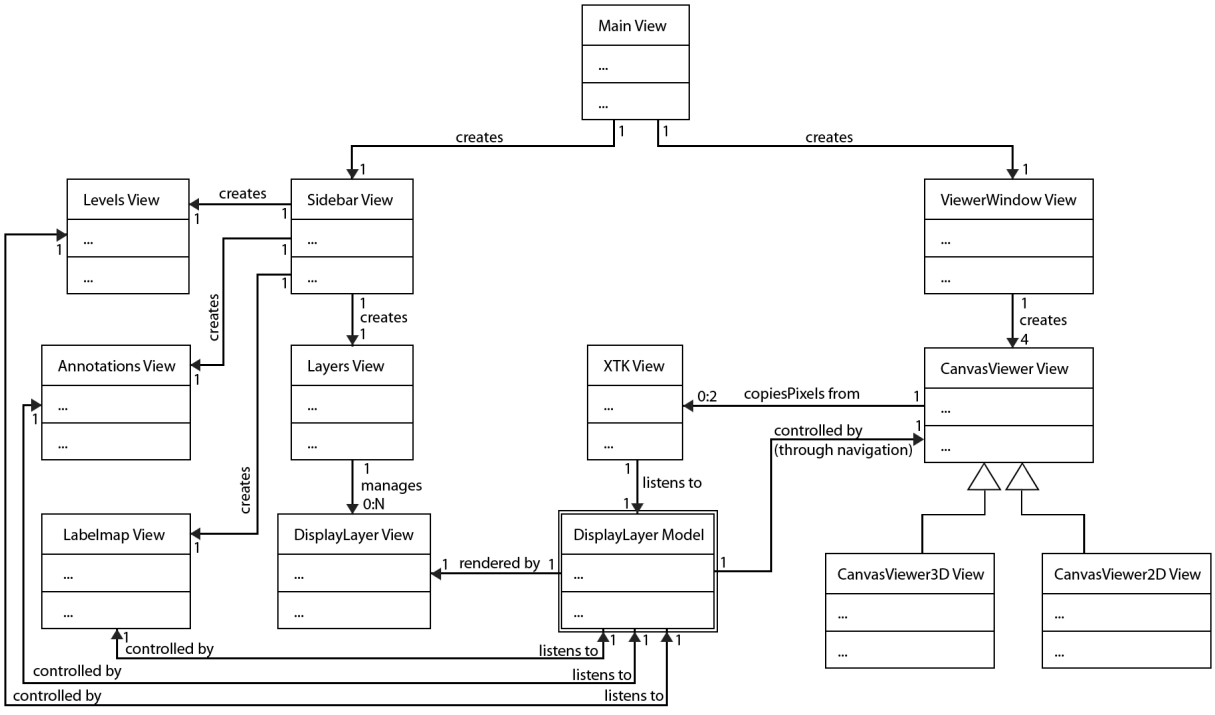


Figure 26: General front end UML Class Diagram

The key idea is that the Display Layer is the main *Backbone* model and thereby central to the whole architecture. It is here that file data is stored for each volume, as well as attributes such

current indices, brightness, windowing and everything else relevant for displaying a Volume File. Figure 27 shows the class diagram for the Display Layer Model.

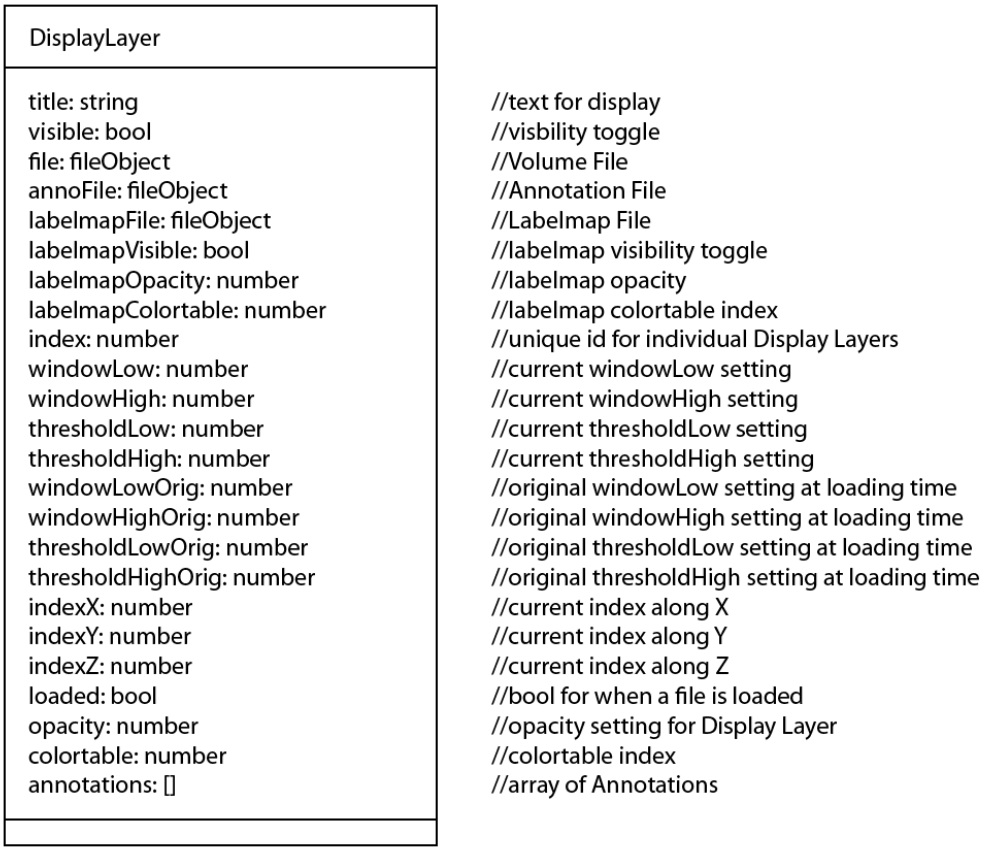


Figure 27: Display Layer Model Class Diagram

Each Display Layer Model instance has a Display Layer vView, which renders a Layer item in the Display Layer Panel. Everything else is managed by *Backbone* Views. These Views generally listen to the DisplayLayer Model to see if any attributes have changed. If this is the case, then the View is updated accordingly due to internal function calls. The individual Display Layer Views are managed by a Layers View (which admittedly could have been named more clearly), which handles Layer Creation, Deletion and Ordering. The specific settings for a Layer, as displayed in the Levels, Annotations and Labelmap tabs are also separate views. Due to the nature of *Backbone*'s Views taking on the role of MVC Controller as well, these Views listen to events from the Model, but can also set attributes of the Model directly, as illustrated in the diagram.

For rendering, there exists a main ViewerWindow View which controls functionality for the four DisplayPanels. Each DisplayPanel is also its own View. In fact, these have an abstract parent View called CanvasViewer, which CanvasViewer2D and CanvasViewer3D inherit from. Importantly, there is an *XTK* View, which contains the actual *X.renderers*. All interaction of the front end with the *XTK* library back end happened exclusively through this View. An instance of this view is created for each DisplayLayer Model. These Views listen to the DisplayLayer Model and update the contained *X.renderers* accordingly. These *XTK* Views are hidden from the user, but their *Canvas*' pixel values are copied by the CanvasViewer Views. When a new Display Layer is selected, the CanvasViewers stop copying the old *XTK* View's canvas and instead copy from the new *XTK* View linked to the new Display Layer. Again, the fact that Views also act as Controllers, means that user interaction with the CanvasViewer View (such as scrolling to set a new slice index) can directly set attributes in the model.

The decision to have the CanvasViewers Views copy the pixel data from the *XTK* Views was important, in that it allowed more customisation of the main View Panels. The image data from the *XTK* Views were essentially one image buffer. I could now add additional buffers as required.

For example I added the Overlay feature, which displays some info text (like current Slice index and filename) in the *Canvas*. Also the opacity feature and Buffer swiping (which will be discussed later) were easier to implement. Technically, the copying is done with Canvas' *drawImageData()* function and the *requestAnimationFrame()* function. *DrawImageData()* allows for drawing on a *Canvas* the contents of an image file or another *Canvas* element (the *Canvas* of the *XTK* renderers in this case). The *requestAnimationFrame()* function is new with *HTML5* and is intended to replace the use of *setTimeout()* which was traditionally used to trigger at specific interval time. It basically aims to deliver the best performance given the current CPU load and monitor refresh rate [2]. I actually tried using a *setTimeout()* method first, where I specified the *drawImageData()* function to run every 50 milliseconds, but it took noticeable more CPU power, which I only noticed when the fans of my laptop started making loud noises. Checking the Activity Monitor, I saw that *ScanView* was taking up more than 50% of the CPU's processing power. By using *requestAnimationFrame()*, this was reduced to less than 30%.

The *XTK* View in itself contains of four *X.renderers* (3D, X, Y and Z). Initially I was planning to separate each *X.renderer* into a separate *XTK* View. However, due to *XTK* implementation, there exist some dependencies between these X.renderers, in that one *X.renderer* is a designated first viewer which takes care of all the file loading and parsing. When this is done, a signal called *onShowtime* is emitted. This is the cue for other renderers to start rendering their newly loaded content, without having to load anything again. Due to this interplay between the *X.renderers*, I decided to bundle all of them into one View, and let all access to the *X.renderers* be handled by the *XTK* View.

Finally there is a MainView which creates a Sidebar View and the ViewerWindow View. The Sidebar View creates the DisplayLayers View as well as the Levels, Annotations and Labelmap Views.

3.5.2 Loading Volume Files

After creating the initial structure, tackling the issue of loading local volume files was the next step. This can be tricky with web browsers, as for security reasons file I/O is not a feature that is readily supported. However I had the *SliceDrop* code to look through. This was all implemented in the XTKView, and basically copying the data into a local container did the trick. This is done with the *HTML5 FileReader* element, which allows for asynchronous file loading in various formats (text/buffer, etc). The *FileReader* loads a file asynchronously and emits a signal when it has finished. The resulting fileObject could then passed onto the *XTK* renderer to load and parse the file.

Once I had this setup I came across one of the first bugs in *XTK*. When loading NRRD files, the volume would load and display, but it would be slightly offset. Testing this on a couple of Desktop solutions which worked fine led to the conclusion that it is a bug in the *XTK* library.

3.5.3 View Panel Layout

In terms of the layout of the components of *ScanView*, it needed to accommodate the four different views (3D Camera View and X, Y and Z Camera Views). I wanted to add some customisation options in terms of size of the viewer and choice of Camera View in each panel. Depending on whether it was a 3D or 2D Camera View, the Display Panel would have different options. The 3D Camera View needed a toggle for Volumetric Rendering, 2D Camera View needed a toggle for Information Overlay.

For the different layouts I briefly investigated using interactive splitters, but found them to be not very easy or reliable to tie into the program. Instead I opted for 4 different static Layouts (See Figure 18). This should give the user enough options to choose what his preferred layout. Implementing the logic behind these panels took a while to get right. Panels had to auto-resize every time the layout was changed. Again this required some exposing of extra *XTK* attributes, as well as *CSS* management of the CanvasViewer Views.

Implementing the feature to change the current viewer orientation took a little bit of time

to figure out. I experimented first with a fixed order of CanvasViewers (meaning their order in the *DOM*), where changing a CanvasViewer's orientation would cause its *Canvas* element to start copying a different *XTK* Viewer. This proved to be cumbersome to manage, as not only the *Canvas*' content but also the buttons of the CanvasViewer had to change (e.g 'Overlay' or 'Volume Rendering' toggle) depending on if the 3D camera view or the orthogonal views were selected. Instead I opted for a floating order of CanvasViewers, so that depending on selection, the CanvasViewer would be reordered in the *DOM* to appear at the correct position. This meant that the CanvasViewer View itself stayed intact with all functionality, only its position and possible size had changed.

3.5.4 Buffer Management

As the next step, I introduced Buffers. Conceptually, this was taken from the Image Compositing Software *Nuke*, discussed in the Desktop Software section, where the user can store different Display Layers in the Buffers. This involved some deliberate thought, as the interplay between the buffers would determine the rendering on them in the CanvasViewer Views (with opacity and swiping). I created a separate *Backbone* Model to keep track of which Display Layers were stored in which Buffer. I then changed the code of the CanvasViewer Views, so that they always store a background and a foreground DisplayLayer. The CanvasViewer View listens to the Buffer Model to determine which Display Layer will be in the foreground, and which in the background. This comes down to the stored Display Layers and the currently selected Buffer.

With the above in place, I could easily implement opacity blending and buffer swiping, since I had access to both Display Layers in my CanvasViewers. I changed the code for the CanvasViewer *draw()* function to draw first the background Display Layer's content and then the foreground Display Layer's content on top of it (only if the opacity of the foreground was less than 100%). For drawing to Canvas with opacity, I made use of Canvas' inbuilt *globalAlpha* attribute. So when using the *drawImageData()* function, it would only draw it with the specified opacity of the *globalAlpha* attribute, thereby revealing the content of the background layer. For buffer swiping, a similar approach was used. Again, the background is being drawn fully, and the foreground is only drawn within a specific region as controlled by the user. This was easily done, as the *drawImageData()* function takes *Canvas* coordinates as input of where to draw to.

3.5.5 Colortables

The next feature I tackled was Colortables, with which to change the color output for a given volume. The *XTL* library does support Colortables, which ironically ended up costing me a week of coding time. *XTK* Colortables follow the same format as *3DSlicer*'s format and a Wikipedia site with which one can create custom Colortables exists online. The format is that each line is prefixed by an index, which will usually by the RGB value of the volume data. Each line is then mapped to a label and a set of RGBA values.

XTK deals with Colortables in that the user has to load them as a .txt file when adding the Volume data and other input files. Thereby it is bundled up with the rest of the IO processes. Again, as with normal file loading, *XTK* is not designed for adding or modifying these kinds of files at runtime, so I had to add this functionality. This took a long time in that I had to trace the whole function flow of *XTK* when loading files. I proceeded with various steps, but would encounter problems at every stage, since I was forcing *XTK* do something that was not really inherent in its initial design. At a very basic level, colored displays (ie varying R, G and B channels) were not enabled for volumes by default, so I had to edit the code for that. On top of that, *X.renderer2D* and *X.renderer3D* deal with color and rendering quite differently. The *X.renderer2D* does the rendering to screen space at a specific time per second. The *X.renderer3D* uses a fragment shader (like in *OpenGL*) to assign color. Figuring all this out took a while, and after 7 days I had finally managed to get the whole process working. The only problem was that since I/O was involved, it would take quite a bit of time to update, so it was not entirely slick. Also, the way *XTK* loaded these colortables was by actually baking the colors into the slice texture.

```
# Color table file E:/Program Files/Slicer3
ces/ColorFiles/PETCT_Labels.txt
# 27 values
0 Background 0 0 0 0
1 R_caudate_head 99 184 255 255
2 L_caudate_head 10 10 255 255
3 R_thalamus 152 245 255 255
4 L_thalamus 0 134 139 255
5 R_frontal_cortex 124 205 124 255
6 L_frontal_cortex 69 139 0 255
7 R_parietal_cortex 124 252 10 255
8 L_parietal_cortex 205 205 0 255
9 R_cerebellum 255 236 139 255
10 L_cerebellum  255 153 18 255
11 R_hippo 205 133 63 255
12 L_hippo 233 150 122 255
13 R_paraventric_WM 255 106 106 255
14 L_paraventric_WM 205 0 0 255
15 CC 255 52 179 255
16 R_olfactory_gyrus 205 105 201 255
17 L_olfactory_gyrus 137 104 205 255
18 All_CSF_space 154 50 205 255
19 R_SVZ 225 245 36 255
20 L_SVZ 242 245 41 255
21 R_Subortical_WM 255 165 153 255
22 L_Subortical_WM 255 187 128 255
23 Total_Counts_in_brain 0 255 51 255
24 R_GM 74 241 233 255
25 L_GM 206 241 179 255
26 Label_26 241 209 234 255
200 BLAH 255 255 255 255
#EOF
```

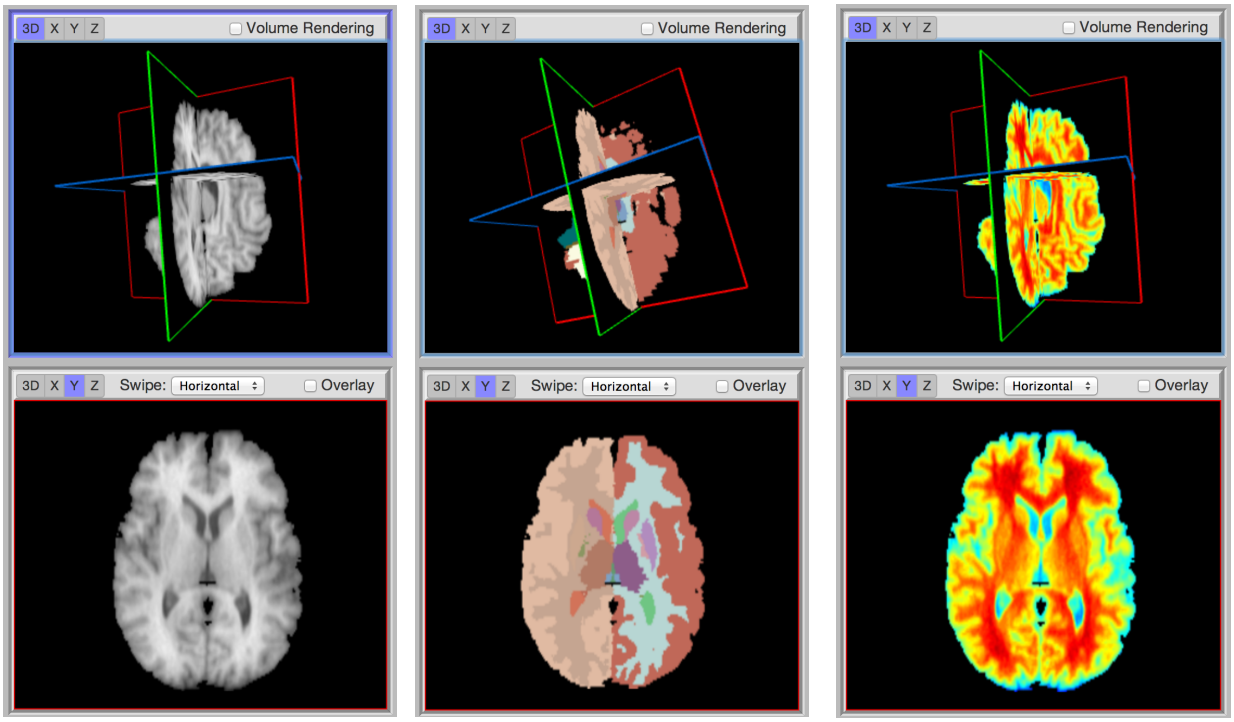
Figure 28: Example of a Colortable

Having finished this feature so far, it occurred to me that I could actually speed the whole process up by having some hardcoded colortables in the code, and using them as a array to look up color values when rendering the actual pixel onto the *Canvas*. This was much easier to implement, loads much quicker as it is more efficient in only manipulating color at a pixel level rather then changing the color values of the whole volume. In the end I chose this method as I preferred the faster feedback. It has the limitation that the user can not load his own colortable, but this is a feature that could be added in future. For the moment, there are three colortables provided. Firstly, the standard colortable maps RGB values directly to the same value. Secondly, an ID map maps values to different RGB values, which is intended for use with labelmaps. Thirdly, a type of Heatmap map has been included which maps the original RGB values from BLUE to RED to mimic the behaviour of a heat map. This was advised by the project supervisor, and an original heat mapping was supplied in a custom format, with only 64 integer mappings provided. I wrote a *Python* script which would convert this into a *JavaScript* array with 256 mappings. I also used a similar *Python* script to convert *3DSlicer* Wiki Colortables to useable *JavaScript* arrays.

3.5.6 Annotations

The Annotations feature was added in the last month and I was aware of the time pressure. That is why initial design decisions had to be made to reduce wasted time later in the process. I decided on implementing rectangular annotations first, since that would make calculations easier to manage. Annotations would be *JavaScript* objects with core members that were predefined (*labelName*, *color*, *3DPoints*), and secondary attributes that would have to be computed on the fly (such as 2D Points for displaying on screen). Specifically, an annotation would be defined by 8 points in 3D Space. However to render it in a 2D View, 2D points had to be generated in screen space. I had intended the 2D points to be selectable in the Canvas View, so these 2D points should also be separate *JavaScript* objects with custom attributes.

An initial UML diagram was designed as in Figure 30, although the final Annotation class



(a) Colortable Option 'None' (b) Colortable Option 'IDs' (c) Colortable Option 'Heatmap'

Figure 29: Colortable Examples

contained quite a few more helper functions to help with the conversion from 3D to 2D space.

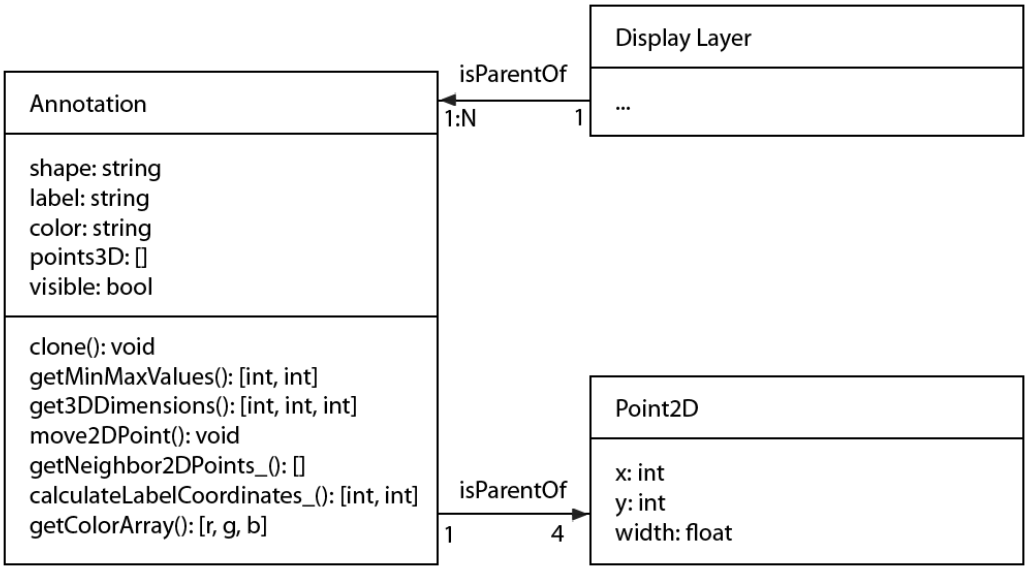


Figure 30: Annotation UML

The next step was to chose a data format and design the workflow. Initial tests were done with *XML* but proved to be somewhat cumbersome since some parsing is required and in the program the *XML* structures would have to be converted to *JavaScript* objects. Therefore the much more intuitive *JSON* format was chosen, since this basically stores *JavaScript* objects in readable form, and can be loaded directly into the program without almost any conversion required. The *JSON* format was originally derived from *JavaScript* which explains its ease of use.

Probably the most involved work was to figure out the conversion of 3D points to 2D screen space. The 2D points would be calculated by each *XTK* renderer separately, since the different

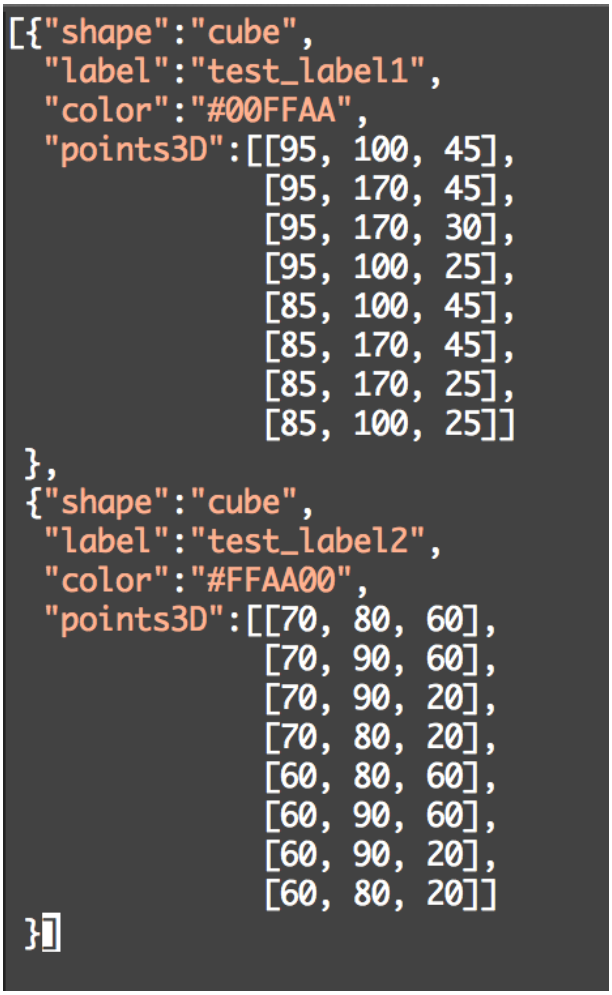


Figure 31: Sample Annotations *JSON* file

views required different 2D screen space points. An array of the 2D points would be stored in the Annotation class. These points would then be updated whenever the position of the 3D points of the Annotation changed. The *XTK* renderer class has a function that translates 2D screen coordinates to 3D slice index coordinates which was a good starting point. I chose a simplified version of this function and implemented a conversion function, which appeared to work fine. I had to develop a much deeper understanding of the Volume File and its various Slice Depth Quantifiers, such as spacing and direction. These had to be integrated into the conversion since they define how the slices are spaced in relation to each other.

When I had finished my naive implementation, some small bugs were noticeable. For example all 2D points of an Annotation would sometimes jump by a pixel even when only one 2D Point was being manipulated. I attributed this to some floating point or rounding errors, but could not track down where the errors were happening. Finally I had to admit that my simplified conversion function was not accurate enough, and I set out to reverse engineer the 2D to 3D conversion from the *XTK* library. This involved some inverse matrix multiplication, and in the end worked perfectly. These changes had to be made in the *XTK* renderer class.

In order for the user to manipulate the position of the Annotation points in screen space, I had to implement manipulatable corner points of the Annotation. This was made easy by the decision to have the separate 2D Point class. Initially I had contemplated creating a separate Manipulator class, but realised that this was superfluous and would just be repeating code. In the 2DPoint class I specified the position and width, so that in screen space I could make the 2DPoints appear wider for easier selection with the mouse. Some internal logic had to be added to the Annotation class, so that when the user changes the position of one 2DPoint, the neighboring points would also transform accordingly, since the shape had to always stay rectangular. I had to revise this is a

couple of times, since in my first naive implementation, there would problems when the Annotation box would collapse so that all four 2D Points would be at the same screen space position. In the end I opted for limiting the size of the Annotation so that the points could never collapse to the same position.

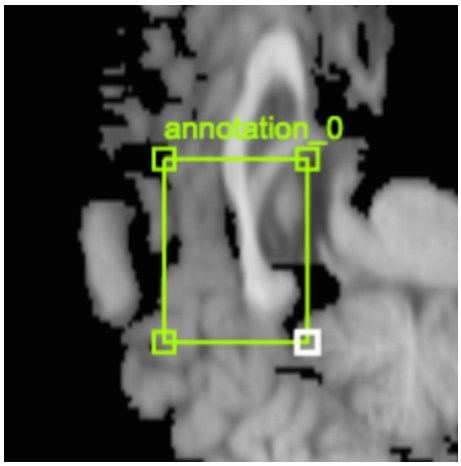


Figure 32: Annotation with bottom right Point selected

An important part of the Annotation Management was the feature to download Annotations as a *JSON* file. However again due to web browser security restrictions, there is no 'Save File As' feature as I had imagined. However after some research, a viable alternative was found in that it is possible to 'download' a file (even if the code is running locally). This was deemed sufficient for this project, even if it does not allow for specifying the file name directly. With more time, it would be possible to add a pop-up window just like a 'Save as' dialog that would then trigger the file download.

Saving files to *JSON* format was straight forward due to the before-mentioned natural overlap of *JavaScript* objects and *JSON* format. The saved files would only include the core information such labelName, color, 3D points and type.

To let the user specify a color for the Annotation, I needed to introduce a color picker. After some research, it became apparent that a host of color pickers were available online and it would be faster to use one of them than implement my own. I downloaded a couple of ColorPickers from the web, and deemed ColPick [29] to be better suited. It has a very basic interface that was deemed sufficient for choosing an Annotation color. Installing and implementing it into the project was also very straight forward.

Once I had implemented the code for creating and editing one Annotation, I created another layer system similar to main Display Layer management. Final tweaks to Annotation management include highlighting the Annotation Layer in the Layer View when hovering over or transforming one of the 2D Points.

3.5.7 Labelmaps

Adding Labelmaps proved to be relatively painless due to the time spent understanding the file loading process when implementing the colortables. The loader had to be forced to refresh when the labelmap was added on the fly. I restricted the program to only overlay one labelmap. I hooked up the opacity and colortables as for the Display Layers. I set the default colortable to chose the ID colortable preset, as that made more sense given the nature of labelmaps.

3.5.8 Controls

Dedicated effort was put into making the software intuitive for users to use, after having researched specific requirements for user input and understanding how many of these files could be inspected in a typical session. Since I was copying the pixels from the *XTK* render canvases into the custom

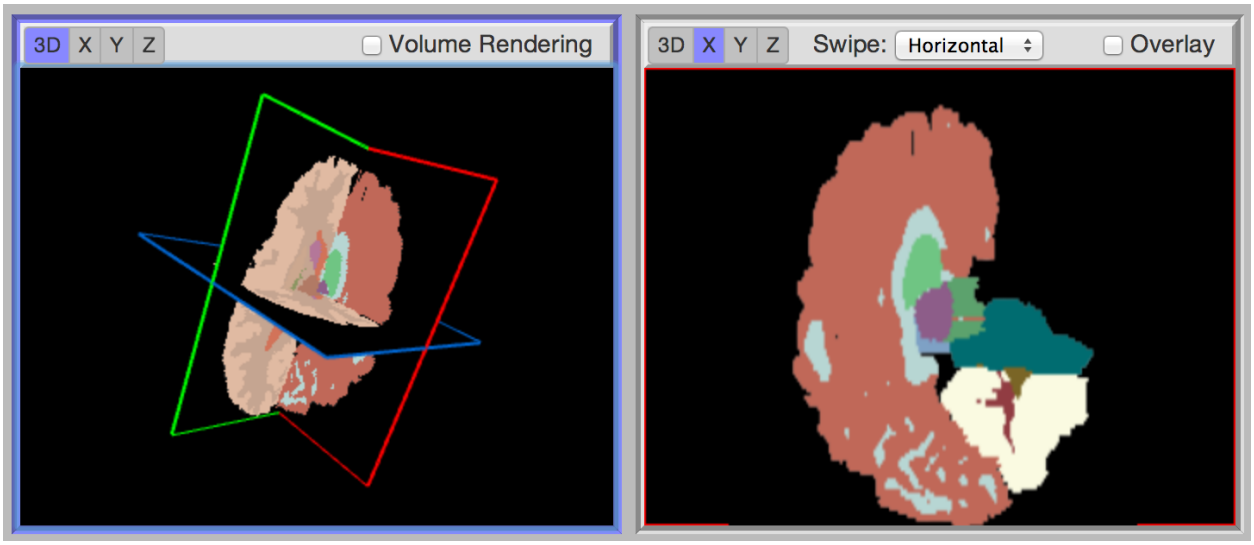


Figure 33: Brain Scan with overlaid Labelmap

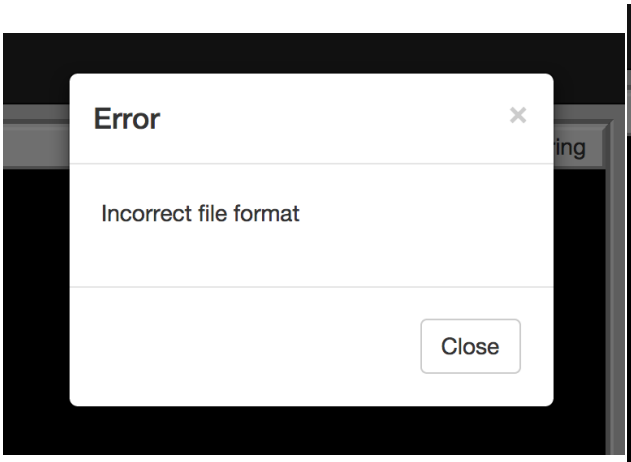
Render Panels, I had to reinstate the controls that were already in *XTK*. However this also gave me some freedom to redesign certain controls.

I scrapped the *XTK* mouse controls for brightness and contrast, since they were buggy, as discussed in previous sections, and there was not enough visual feedback for the user to see what was being changed. Instead I opted for traditional slider based methods, with numerical input fields on either side to clearly show which numbers were currently set. This works well in that it is very clear to the user how to manipulate the image, and it is more bug-free than the *XTK* implementation. Also the number input fields allow for precise control for the case that a user wants to recreate a certain brightness configuration.

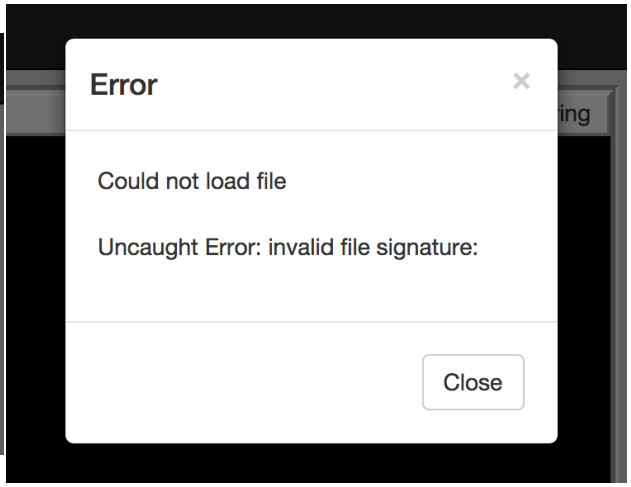
All slice navigation and traversal was placed on mouse controls. The guidelines from the book helped inform the controls in that radiologist prefer to have as simple controls as possible since their work can consist of dealing with vast quantities of these kind of files, so any intricate control input method would soon get tiresome. Therefore the standard *XTK* input methods were kept mostly intact, so the middle clicking pans the camera, right-clicking zooms and left-click rotates in the 3D View. Since the left-click for the 2D Views was freed from controlling brightness, I assigned the traversal function to this, which allows the user to quickly reset all 3 slice indices. This was inspired by *Papaya* and preferred to *XTK*'s default traversal input which requires the additional holding down of the 'Shift Key'.

Reinstating the mouse controls took some as I had to find out how to trigger the appropriate response in the *XTK* canvases. For 2D and 3D navigation, I had to study the *XTK* X.camera more closely. It turned out that I could access the *X.camera*'s view matrix directly via an already implemented setter method, where indices 13 and 14 were synonymous with the X and Y position camera, and index 15 was the Z position. Therefore all I needed to do was calculate the relative movement to the currently selected *HTML* Canvas (by substituting the absolute mouse coordinates from the *HTML* Canvas coordinates), and convert this with an appropriate factor to X and Y numbers to be entered into the view matrix. Likewise for zooming in the 2D Views, an adaptive zoom factor was implemented after discussion with the supervisor. The first normal implementation would zoom with a constant factor, but this would be more ineffective the further the camera was zoomed in on a slice. I implemented an adaptive zoom factor that interpolates between a linear zoom function to a square zoom function to counteract this effect.

To further add convenience for user input, I created a couple of keyboard shortcuts. To toggle between the A and B buffers, the user can press '1' or '2'. I was used to this from using the compositing software *Nuke* and have always found a frequently used short cut when comparing two images with each other, especially when looking for minute differences. This ties in with the idea that *ScanView* could be used to compare two medical image data files with each other. For



(a) Error message for incorrect file format



(b) Error message for specific load error

Figure 34: Examples of Error Messages

the View Panels I also added the keyboard 'f' shortcut to recenter the camera after any panning, zooming or rotation had been applied, inspired by *ITKSnap*'s image centering feature. This is achieved by storing the initial camera view matrix when the file is loaded, and loading this view upon key press. Adding these keyboard shortcuts is relatively trivial and more could and should be added, assuming that keyboard short cuts increase user speed. Some care had to be taken to ensure that the keyboard shortcuts do not get triggered when a text field is being edited.

3.5.9 Error Handling

An issue that is entirely related to the *XTK* library is how to let the user know that a certain file has not loaded. This was an important issue since a large variety of files downloaded from the Internet could not be opened with the *XTK* library. Various errors would be thrown in the loader.js and parser.js components of the *XTK* library, but there is no simple way to propagate this to the higher API-using layer, since the file loading is handled asynchronously and with event sending. There are no events implemented that signal the failing to load a file (other than failing to download a file via HTTP). Due to limited time, rather than adding these events into the *XTK* library, I added an error listener to the UI Layer which would call an errorHandler function whenever an error was thrown. This errorHandler function would then do a string comparison and cause a pop-up modal window to appear to inform the user of the specific error. This had the benefit of being able to screen for only certain types of errors.

3.5.10 Slice Animation Feature

This feature was added fairly late in the project, but it was relatively fast to implement. It had been suggested in (Preim & Botha, 2014), and when talking to a radiologist, it seemed like it could be a useful feature which would relieve the user from manually having to scroll through a Volume file. I implemented this via a set of 5 buttons from the *Twitter- Bootstrap* library, commonly used to signal playing, stopping or rewinding a media file. I used the *setTimeout()* method to recursively increment or decrement the current slice index (dependent on the viewer). At each recursion, it checks a guard variable whether to keep looping. The stop button changes this guard variable to on and therefore breaks the loop. I also trigger this break when the user uses the mouse scroll button, as I figured that the user might like to stop the animation easily other than by hitting the stop button. Almost by accident, it also provides the user a way to quickly navigate from to the minimum and maximum slice numbers for each direction by use of the fast-forward and fast-backward buttons.

3.5.11 Polishing the User Interface

Now with the program gaining shape, it was time to tweak the User Interface. This included the not very glamourous tasks of making sure that any Display or Annotation Layers would never be bigger than their container element by adding scroll bars. Luckily, this is a *jQueryUI* feature that is easy to implement, and one can specify along which axis one would like the scrollbars.

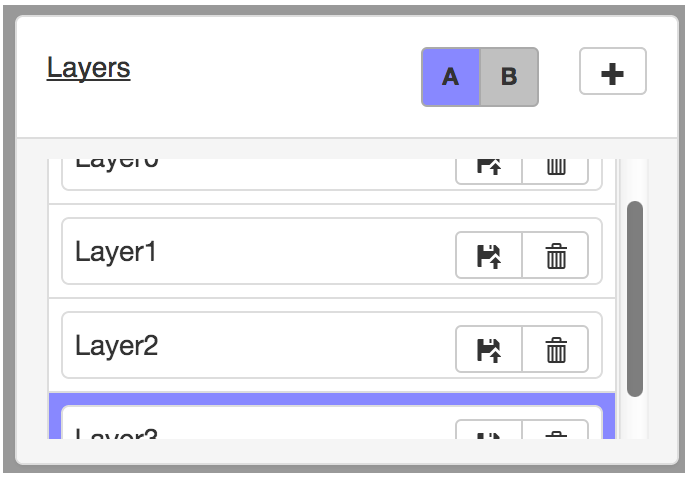


Figure 35: Vertical Scrollbar for Display Layers

Furthermore, I implemented changes that would guide the user better in terms of using the software. This included disabling of all input and buttons in the Levels, Annotations and Labelmap tabs when no appropriate medical imaging file had been loaded yet. Also I added pop up windows to warn a user if the wrong kind of file was being loaded. Tooltips (from *Twitter-Bootstrap* library) were added to almost every button to help the user understand the intended functionality of each.

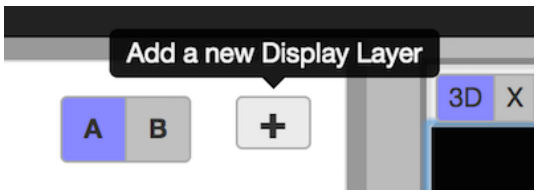


Figure 36: Twitter-Bootstrap tooltip example

3.5.12 Deployment

In order for users to be able to evaluate the product, some effort had to be put in to deploy the software to the web. I am hosting it on my personal webspace, which I preferred to hosting it on a local college server since I can send the site for feedback to people outside of Imperial College. I use *FileZilla* to upload the code, but wrote a *Python* script to facilitate the uploading, since manual replacing of files soon became tedious and time-wasting. The script will recursively copy each file of a specified directory to a designated space on my web space.

The deploying process went smoothly, as it had been designed as a live webpage from the beginning. I added a page with some sample data for the user to download, tutorials resembling the protocols above together with screen capture videos. Furthermore an 'About' page was added which explained the main features of the program as well crediting various libraries and plug-ins that were used, as well as patch notes.

Finally, an online survey by *Survey Monkey* was introduced to allow for convenient feedback by the user. *Survey Monkey* allows for free creation of surveys that are hosted online and accessible via a web link that can be sent by email or in this case embedded into a website. They also offer a host of analytical tools for investigating survey answers and data. The more advanced tools need to be paid for, but even the free tools allow for summary of results and inspection of individual

responses. In lack of a proper beta test, this was deemed a quick way of getting feedback from different users. It was deemed important to ascertain whether the user has some experience with medical image data and viewing software, how much time the user spent with *ScanView* in order to estimate how thorough the usage was, which bugs were found and which features could be improved or added, The following questions were asked:

1. Have you worked with medical image data before? (Yes/No)
2. If yes, in what capacity have you worked with medical image data before?
3. Have you worked with medical image viewing software before? (Yes/No)
4. If yes, please states which programs you have used!
5. How much time did you spend with the ScanView program? (0 - 15 Minutes, 15 - 30 Minutes, 30 - 60 Minutes, 60 Minutes and more)
6. How convenient is the *ScanView* program to use? (Extremely convenient, Very convenient, Moderately convenient, Slightly convenient, Not at all convenient)
7. Do you think the *ScanView* program could be a viable alternative to comparable non-browser-based Medical Imaging Viewing software (eg. Slicer3D)? (Agree strongly, Agree somewhat, Not sure, Disagree somewhat, Disagree strongly)
8. Do you have any criticisms of the *ScanView* program?
9. What features would you like to see added to the *ScanView* program?

A host of fixes have been uploaded since initial deployment, partially fixing outstanding bugs and implementing features requested from user feedback. More on this topic will be covered in the results section.

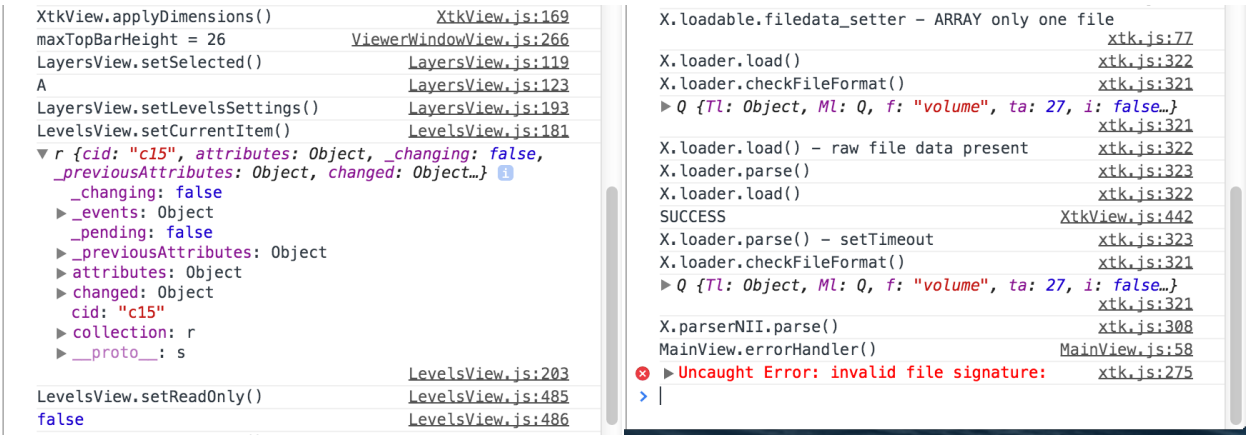
3.6 Implementation Details/ Working with Libraries

3.6.1 Version Control

To keep track of my project, I used *Git*. This was an obvious choice since it has been the DOC-preferred version control system as well as *XTK* being hosted on *Github* which I had to fork. I managed my own project code in a separate Version Control from my *XTK* fork in order to not confuse the two. *Github* proved as per usual very useful and reliable in tracking down changes from earlier versions as well keep submodules up to date for the *XTK* library.

3.6.2 Browser Developer Environment

Initially I started developing in *Chrome* since it was my primary target for running the software. A few weeks later my laptop broke due to accidental damage and I had to work in the Imperial College labs, where I had problems running *WebGL* in *Chrome*, so I switched to *Firefox*. I soon found however, that Firefox leaves a lot to be desired for developing. The most important part of developing for me was the web console. With *JavaScript* it is possible to print to this console, which I used endlessly. It also prints when a runtime error has occurred, complete with line number. This works great in *Chrome*, but in *Firefox* (on *Ubuntu)* at least, the line number would often be obscured. Any another problem is when printing a *JavaScript* object to the console, it shows the final state of this object at the end of the code runtime. This means it is not possible to check the state of an object during runtime, only at the end. This caused some confusion until I had figured this out. *Firefox* is supposed to have a lot of developer plug-ins that can be downloaded, but in general I found *Chrome* to be much more convenient to use, and after a month managed to get a new laptop and returned to developing in *Chrome*.



(a) Chrome console showing *JavaScript* Object (b) Chrome console showing error message

Figure 37: Examples of Chrome console displays

Chrome offers a variety of developer tools that come with the browser. First the 'Elements' tab provides the interactive *HTML* source code of the current web page. When hovering over a line of *HTML*, the appropriate element in website lights up. It also shows the current *CSS* rules that are being applied, with the option of toggling them on or off and getting a live preview. The 'Elements' tool is extremely helpful when debugging if the correct *HTML* correct has been written into the *DOM* or to check which *CSS* rules are currently active or have been overridden.

The next developer tool is the 'Network' tab. This shows the external libraries and source files are being loaded. It gives a good chronological break down of which library loaded when and how long it took. This proved useful in the beginning to see if the dependencies were handled correctly by *RequireJS*.

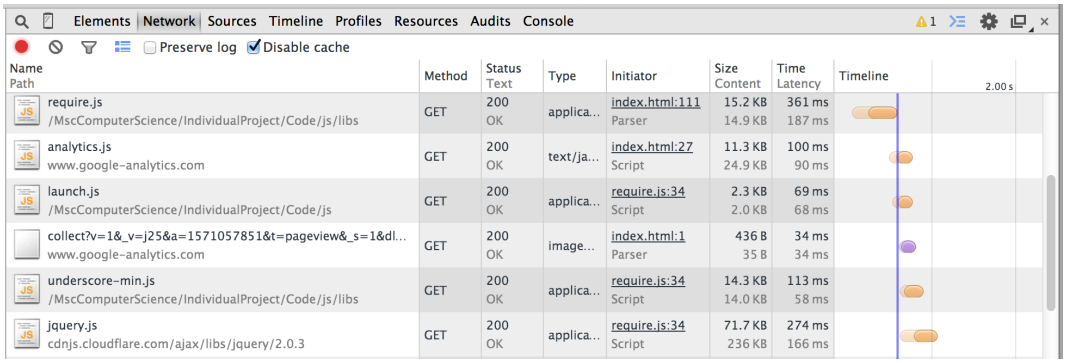


Figure 38: Chrome Networks Tab

Chrome furthermore offers a debugger complete with break points, so it's possible to follow the flow of code by individual lines or break points. This was used sparingly, as when working with *XTK*, the number of files that are being involved is huge and it would take too long to actually follow the code. Nonetheless it was helpful when I had to develop a deeper understand of the flow of control for example during when a file is loaded with *XTK*.

As there was some switching between web browsers, I noticed that *Chrome* and *Firefox* behaved differently sometimes. For example the layout would look drastically different in *Firefox* compared to *Chrome*, sometimes for reasons that were hard to track down due to large amount of variables in play. It appeared as if the *CSS* float command behaved differently, and place items to a different rule set. At other times, mouse navigation would not work in *Firefox*, since somehow mouse clicks where registered differently from *Chrome*. During the project, I spent time trying to bring up browser compatibility for both in parallel, but in retrospect I probably should have focused solely on *Chrome*, and then spent time making it work in *Firefox* and all other browsers, since they will also need custom work.

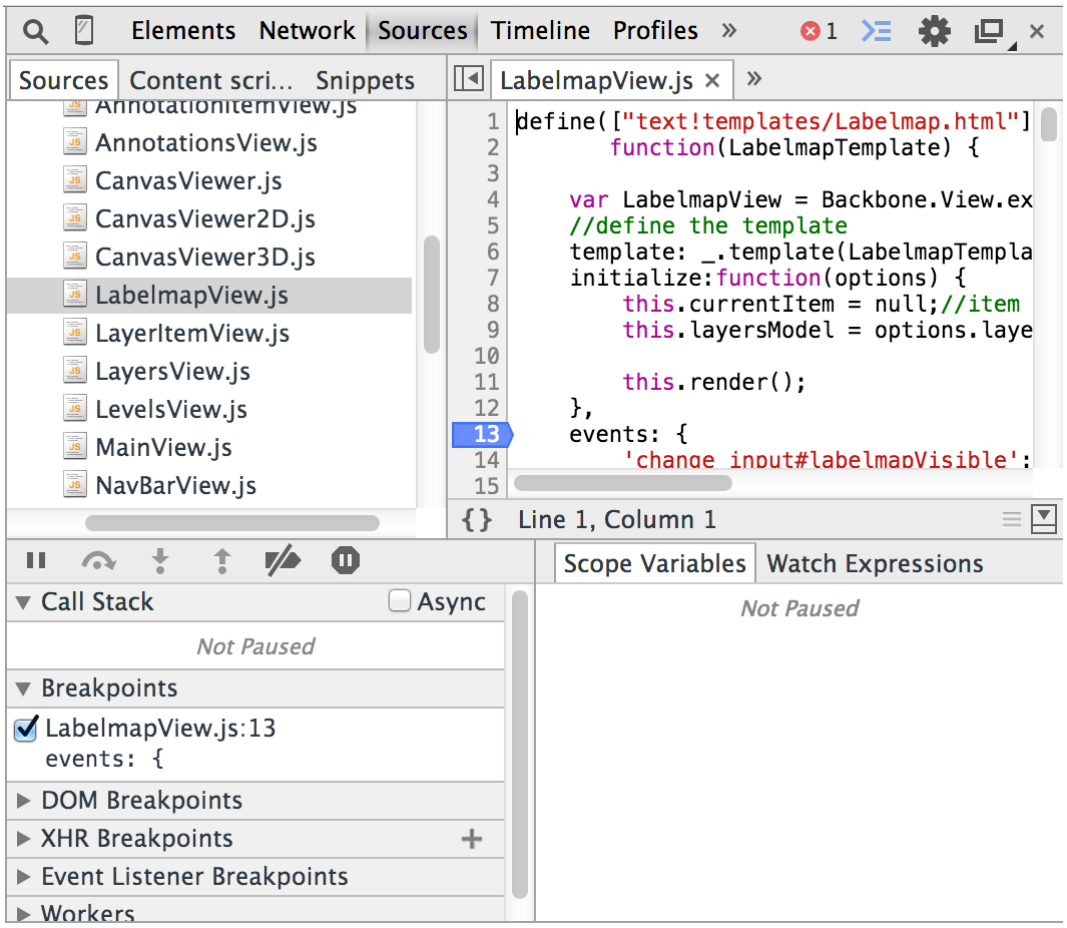


Figure 39: Chrome Sources Debugger with one Breakpoint set

3.6.3 JSFiddle

Related to browser development, a tool I used a number of times was *JSFiddle*. This tool allows to on the fly testing of *JavaScript*, *CSS* and *HTML*. In fact, when asking for web developlement related questions on the StackOverflow forum, answers are often posted with a working *JSFiddle* example. To create an example, one only needs to open *JSFiddle* and one can start coding. There are separate input windows for *JavaScript*, *CSS* and *HTML* code. It also features a windows that displays the resulting output. The link to this code can then be saved or sent to other people. All of the *XTK* tutorials are written as *JSFiddle* examples which allows for quick experimentation and testing of features. This was a really useful and convenient tool.

3.6.4 XTK

As already mentioned, the initial idea was to use *XTK* as an API to handle all the low level work to with loading and displaying Volume Files. As it soon became apparent, I had to dig deeper into the *XTK* library. Since *XTK* uses the Google Closure library to minify the code, I first started to introduce print statements into the code in order to follow the flow better. However this became tedious quite fast and I figured out that is possible to run the uncompiled code with a few steps. This proved to be necessary since I was getting lots of errors at later stages of the project, and it let me actually step through the code with the *Chrome* debugger in a meaningful way.

I needed to write various setters and getters for attributes that I needed to get direct access to, which was trivial to implement. I tried to follow the style guide which is posted online in using camelCase typing with *Underscore* s at the appropriate places for attributes and functions.

When implementing the interactive loading of color tables and labelmaps, I was forced to get a deeper understanding how the file loading process works. The *Chrome* debugger did help with this, and due to the complexity and the number of files involved, I had to draw diagrams which

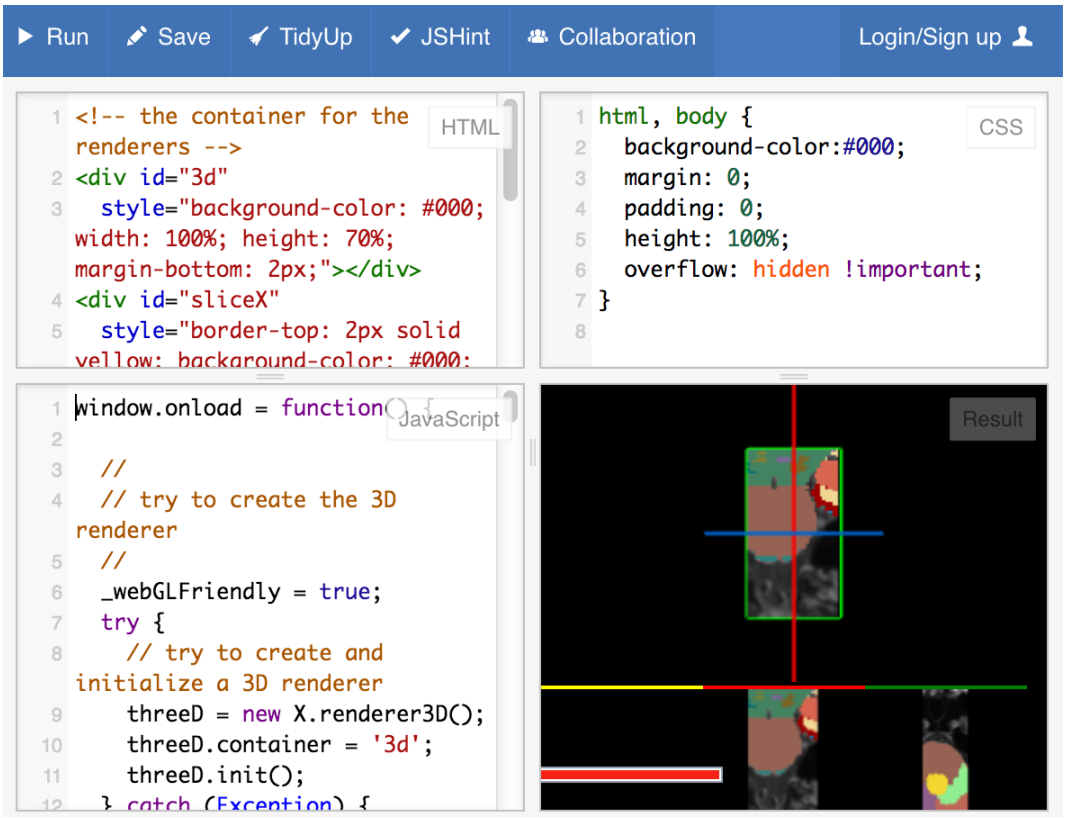


Figure 40: XTK Tutorial in JSFiddle Format

I would consult often. See Figure 41 for flow of control when loading a file for an *X.volume*. The flow of control is the same for all file objects (including colortables and labelmaps), since the update() function steps through the *X.volume* and its children recursively to see if any file needs to be loaded.

In the example, a filename is assigned to the *X.volume*'s file attribute. The *X.volume* is then added to the renderer (1) and the *render()* command is called (2). The *render()* call starts the continuous frame updating at 60 frames per second. At the same time, the *X.renderer.add()* function calls the internal *update()* function (3), which sets up event listeners for the volume. The overloaded *update()* function in *X.renderer2D.js* (4) gets the loader attached to the renderer, and calls the *loader.load()* function (5). The internal *parse()* function is called (6), which in turn determines the required parser (*parserNII* in this case) and calls its *parse()* function (7). At the same time, *loader.parse()* sets up an event listener to wait for a 'Modified' signal which will call the internal *complete()* function. *ParserNII.js* deals with the file format specific loading of data, and employs the *reslice()* (8) and *reslice2()* (9) functions to compute the current slice information of the volume. These are called three times - once for each orthographic view directions. Control is returned to *parserNII.js* after *reslice2()* has finished (10). Then an event signal is sent to *loader.js*, which triggers *loader.complete()* to be called (11). It also sets the volume's *dirty* flag to be true (which, among other uses, is used as a check by the renderer's *render()* function). The loader calls the *modified()* function in *X.volume.js* (12), which in turn calls its internal *slicing_()* function (13), again once per orthographic view. In this last function, the current slice is set to be visible if it has been computed previously in the *parser* and *parserNII*. The *slicing_()* function also gets called at other times, and if the current slice does not have any texture information loaded, the parser's *reslice2()* method is called from here for computation. *X.renderer2D*'s *render()* method which has been running continuously throughout this time, will recognise that the slice number has changed (from null to a specific integer number) and force a redraw of its pixels, thereby drawing the current slice texture to the screen. The *render()* function also checks if any dirty flags have been raised and will force a redraw accordingly.

If this process is being called for the first time, the modified volume will also send an Event

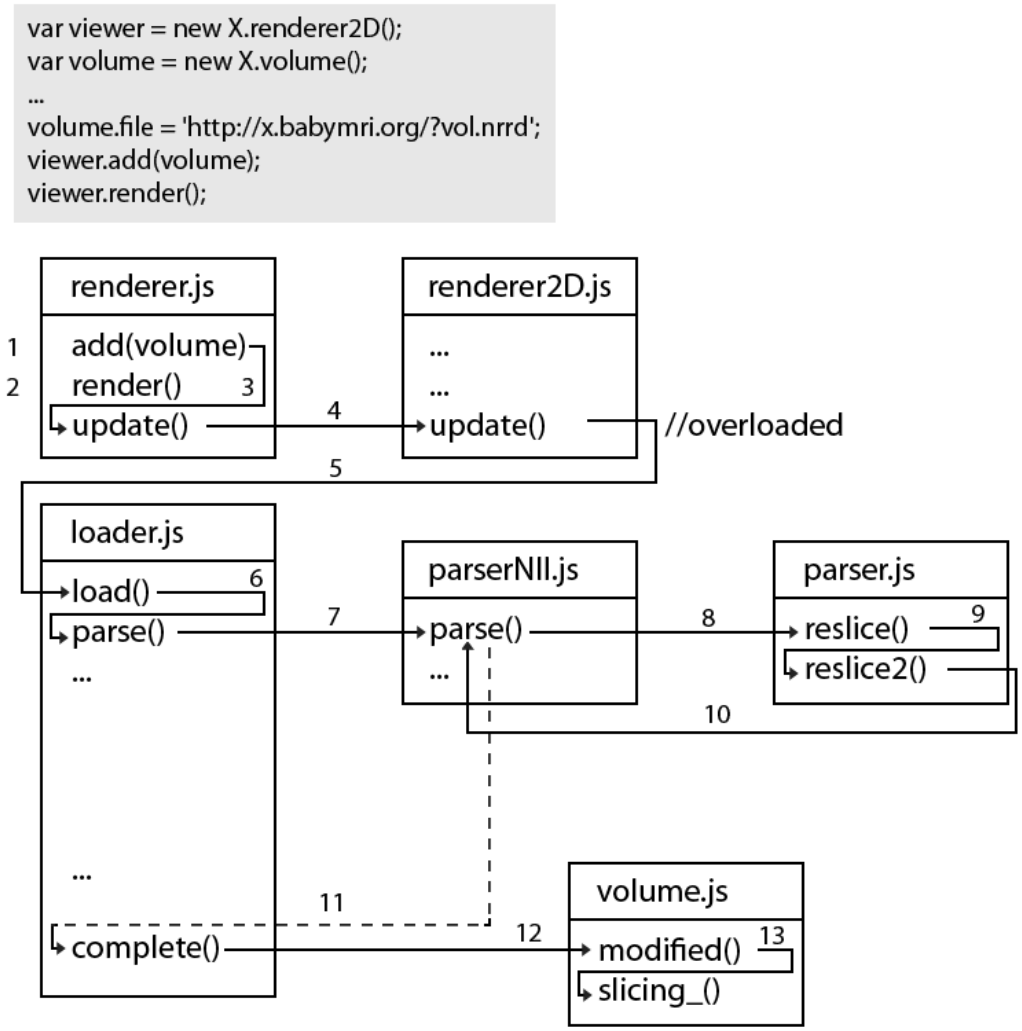


Figure 41: Flow of Control during *XTK* file load

Signal to the renderer that is has been added to, forcing the *X.renderer.onShowtime()* function to be called. This is supposed to be useful when having more than one *X.renderer*, in that one renderer can do the initial loading of files, and then the other renderers can start rendering immediately. Personally I dislike this method, since it forces an interdependence on the viewers and I would have preferred separating the loading logic out from specific *X.renderers*. When asking about this on the StackOverflow forum, one of the *XTK* developers agreed that the current solution is not ideal.

For the *X.renderer2D* class, when the user changes the index of one of the slice directions, the *update()* function is called and the new slice texture calculated. This slice texture is then stored in a local buffer attribute in the *X.volume* object, called *children* so as not to have to recalculate this slice when the user returns to this index at a later point. The *X.renderer.js* object itself is running at a specified frame rate, and will update if it detects that one of a number of things has changed in the volume. Slice index changing is one of them. It was this that caused some problems when causing a redraw after reapplying a colortable or a labelmap. All the current checks would not cause a redraw, since I had not changed the slice index, but rather data that was affecting the current slices. So I had to find another way of triggering this redraw. In the end I opted for querying for the *X.slice*'s *sliceId* attribute, which is a unique identifier number that gets changed every time a slice is recomputed. This way I could finally register the update. Luckily for the *X.renderer3D*, I had no issues, since it just renders the current *X.volume* in the current state, and the computation on a slice index change is being done by the *X.renderer2D*.

An issue with loading colortables was that by default *X.volumes* only get rendering as a single

color channel, and the not all three RGB channels. I found the code for this in the *X.renderer2D* class, where on each iteration of the *draw()* function, only the G channel would be converted into on screen pixels. Changing this to include the R and B channels was trivial. For *X.renderer3D*, this was a bit more involved, since the color management actually takes place in the fragment shader as per the *WebGL* standard. Once I had realised this, it was easy to change the *GLSL* code to support all three RGB channels.

I found a few issues and bugs with *XTK*. One of these is that some files do not load properly, so more advanced file handling should be added. To alleviate this problem, I implemented some advanced error catching methods with warning pop-ups in the UI, so that the user at least knows that something has gone wrong. Without this, it would have just printed out an error message to the web console. Another bug is that the mouse controls for threshold and window setting work in a way that is not intuitive and not reversible, so that after left-clicking and dragging a few times, the image will not return to its original brightness anymore. This could be because there are two attributes (min and low) to set for both threshold and windowing. I worked around this by reassigning the window and threshold controls to the slider and input fields in the Levels Tab.

3.6.5 RequireJS

RequireJS proved invaluable for determining correct dependencies for loading *JavaScript* libraries. As I am using more than a handful of libraries which are also dependent on one another, I needed the ability to ensure that they had loaded correctly. The issue is that the modules are loaded asynchronously, so a module that depends on a another module could easily load before the other (as opposed to in sequence loading from function programming). A good example for this is the tooltips that I implemented towards the final stages of the project. Both *jQueryUI* and *Twitter-Bootstrap* provide tooltips. In both cases they are initiated by the same function call, however they look quite distinctly different. I wanted to use the *Twitter-Bootstrap* tooltips as they seem more aesthetically pleasing. Without determining the load order with *RequireJS*, whenever I reloaded the software a different tooltip style would pop up, depending on which library loaded first. This kind of race condition is of course undesirable.

Another example is of course *XTK* itself which relies on *jQuery* and *Underscore* JS. *RequireJS* really was a life saver for these situations. The screen shot in Figure 42 shows the ordered nature of loaded modules, using *Chrome*'s Network tab. Without *RequireJS*, the loads would happen a lot more randomly.

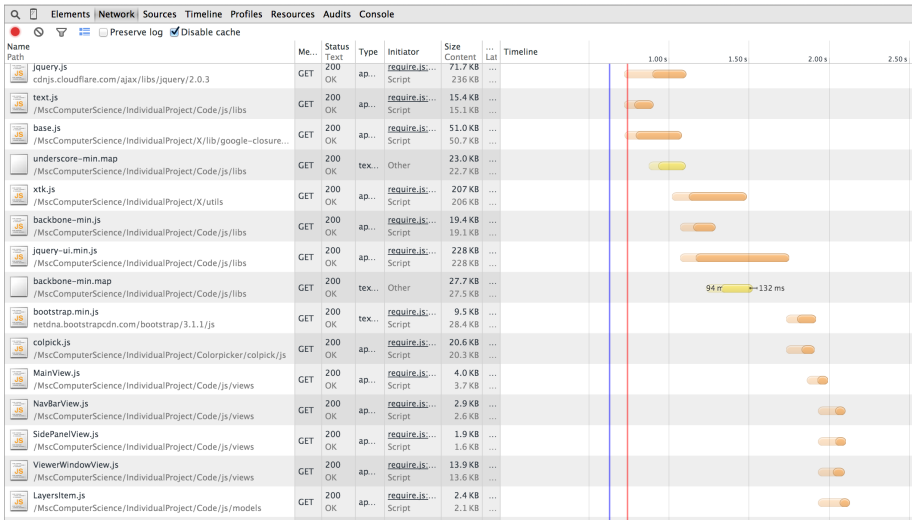


Figure 42: Ordered module loading with *RequireJS*

3.6.6 Backbone

Working with *Backbone* worked well after a steep learning curve in the beginning, especially because I had to set it up in parallel with *RequireJS*. I sometimes was unsure of whether to follow the event-based design or normal functional design. It is tempting to communicate between components via events, but I found it also makes it a lot more difficult to debug, since it's not easily visible which pieces of code are listening to an event that is fired.

Another issue that came up was that I was tempted to repeat class attributes in the Views as well as the Models. This took some discipline and time to get used to. I also had the feeling that in certain situations, the pattern of having Views and Models became a hindrance, since I needed to manage both. In the example of having several Display Layer Models and the Display Layer Views, I found that I had to manage both of these sets of objects. This felt unintuitive and maybe it was due to incomplete understanding of how *Backbone* works. Another case of finding Backbone's functionality superfluous came up with *Backbone*'s Collections. These are a type of array and can be used to store models and also send signals when the contents of a Collection is changed. However this seemed like overkill in the case of Display Layers, so I switched back to using normal *JavaScript* arrays.

One final problem came up with arrays and the *change* event. When changing the contents of an array, the *change* event does not get fired. This is because the base memory address of the array does not change. In order to work around this, I had to clone the arrays every time I wanted to change their contents. I ended up packaging this sort of behaviour in classes that had arrays as members. This issue came up predominately with Annotations, since Annotation objects are stored in a array structure in the Display Layer Model. Also Annotation objects contain arrays of 3D and 2D points.

In general, I found *Backbone* to be essential in structuring my code, and managing the interplay between the internal data and the separate management of rendering *HTML*. I get the feeling that I have still a lot to learn about *Backbone* and would like to spend time to understand the subtleties of this library.

3.6.7 JQueryUI

Generally speaking, for any *HTML* page element, both the look and the functionality have to be considered, as implementing them can be quite complex. This is why a lot of libraries have been created that provide generalised elements such as buttons, file loaders and more complex UI elements. A few like *jQueryUI* and *Twitter-Bootstrap* have been become very popular and have been used in this project. They have the benefit of providing both the *CSS* style definitions as well as predefined *JavaScript* functionality for the elements.

Some features from this library were utilised, such as the tab functionality for the 'Levels', 'Annotations' and 'Labelmaps' tabs. Also, the sliders for setting Brightness, Threshold and Opacity were taken from this library, as they provided a convenient solution both for the design and for the functionality.

3.6.8 Twitter-Bootstrap

This library was used to style the webpage consistently and to make use of their icons, buttons and tooltip functionality.

The library provides a host of useful icons and button presets, such as button groups which were used when grouping buttons close together like the 'Load' and 'Delete' button for each Display Layer. This added a more sophisticated look to the software without the need for individual styling of buttons by writing custom *CSS* classes.

Adding the tooltips proved more involved than initially thought. For dynamically added *HTML* elements such as the Display Layers, custom tooltip options would have to be used and set in order to make them work correctly. So instead of having a general purpose solution, custom settings had to be set for each tooltip. Also, since *Twitter-Bootstrap* and *jQueryUI* both provide tooltips that called by the same initialisation function, sometimes errors would appear in the browser console,

stating that a conflict had been caused. Fortunately, *jQueryUI* allows for downloading a custom build where the tooltip feature has been disabled.

Furthermore, creating modal warning screens when the user attempts to load a file with the wrong extension was made very easy with Twitter-Bootraps modal screen. Again, styling this manually and creating the logic behind it would have been very time consuming.

Twitter-Bootstrap supplies a host of interactive UI elements such as buttons, tooltips, pop up windows, tabs, navigation bars, progress bars and more. It is a widely used library - in June 2014 it was the No.1 project on GitHub with 69,000+ stars and 25,000+ forks (Wiki).

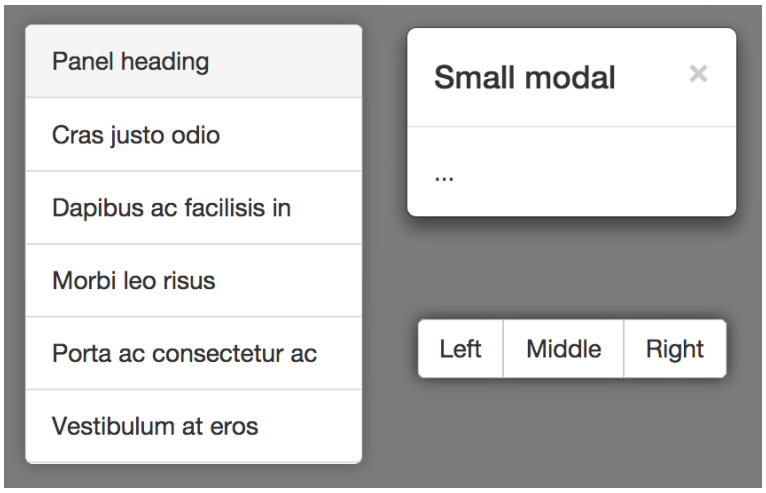


Figure 43: Examples of *Twitter-Bootstrap* components used. Panel (left), Modal (top right) and Button Group (bottom right)

All things considered, this library proved to be very useful and saved a lot of time.

3.6.9 Google Analytics

Google Analytics is a tool by Google that provides detailed information of website usage. I installed it to get a clear picture of how many people would be using the website. I have used Google Analytics in the past and have been impressed by the comprehensive overview it provides. It shows the number of unique and returning visitors, average session duration, feedback on which browsers and operating systems they were using, which city and country it was viewed from. Additionally for mobile devices it even shows the screen resolution.

Implementation is very easy by just copying an *HTML* script code snippet and pasting it into any website that the user would like to track the usage for.

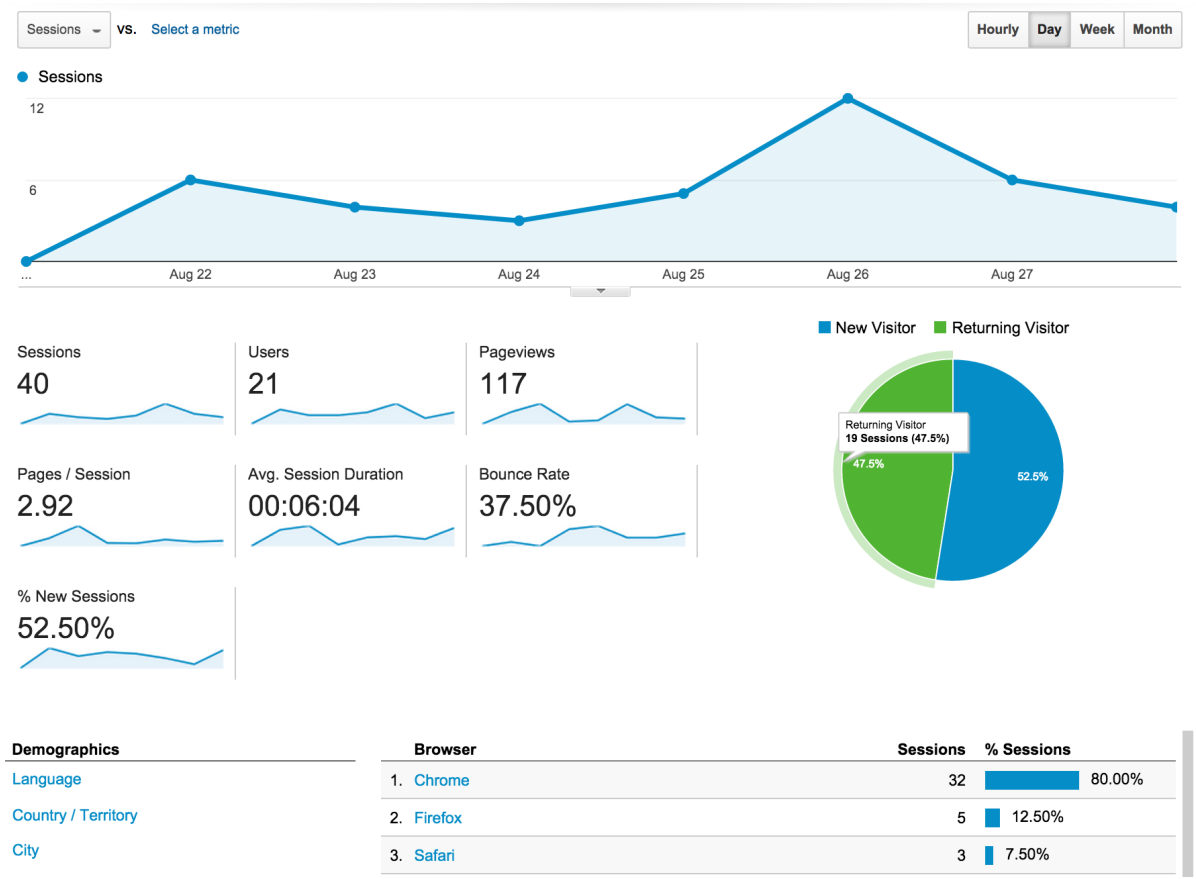


Figure 44: Google Analytics Website Usage Statistics

4 Results and Evaluation

In order to evaluate the software, a number of benchmarks were chosen. First, the software will be evaluated against the aims, requirements and use cases set forth at the beginning of the project. It will also be compared to other browser-based and desktop-based medical image viewer applications. Next, one of the main goals of this project, namely that the website runs on a variety of computer setups, will be tested and analysed via a set of protocols. Finally, as the software has already been deployed on the Internet, user feedback will be discussed.

4.1 Review of Features

4.1.1 Comparing to initially set requirements

When comparing the final features of *ScanView* with the list of initial requirements, it appears that all of the primary requirements have been implemented successfully.

Regarding secondary requirements, only a few have been implemented. This is mainly due to lack of time and complexity of some of these requirements. In retrospect it is much more clear that some secondary requirements are harder to implement than others. For example adding Measuring Tools would be quite easy, since a lot of the basic functionality that would be required is in place, such as conversion from screen space to volume space. Sharing data online would be much more demanding to implement, since it would require dealing with network and could be subject to an entire other project (even though an implementation via DropBox would be feasible).

Primary Requirement	Implemented
Loading NII file	Yes
Viewing file in 2D and 3D	Yes
Navigation in 2D and 3D	Yes
Brightness and Contrast controls	Yes
Loading NII Labelmaps	Yes
Loading additional NII files for comparison	Yes
Different colortables provided	Yes
Sample Data Provided	Yes

Secondary Requirement	Implemented
Paint Custom Labelmaps	No
Save Labelmaps to computer	No
Create and save Annotations	Yes
Method of sharing data	No
Image filters	No
Measuring Tools	No
View volume file as 3D model	Yes
Animation along axis	Yes

4.1.2 Comparing with other web browser based programs

In terms of the final feature list, *ScanView* distinguishes itself from the currently available web-based medical image viewer applications. As most online programs are very basic, *ScanView*'s features outnumber them.

The software that it resembles most closely is *SliceDrop*, also in part due to its common heritage. *ScanView* improves on *SliceDrop* in a number of ways. *ScanView* is more dynamic in that it allows interactive file loading and deletion similar to a standard desktop software program. It allows to load several Volume Files (via Display Layers) which can be compared visually due to the implemented Buffer System. Furthermore file load errors are handled more completely by letting the user know the error message via a message pop-up. In *SliceDrop*, file load errors are only printed to the developer console, which the standard user can not be expected to check. Both *ScanView* and SliceDrop enable volumetric rendering, since this is a common feature of the *XTK* library. *ScanView*'s View Panels are more free to display custom information via the Overlay toggle. In terms of navigation, both feature the same set of transformations. *ScanView* adds the convenience feature of resetting the view camera to the start position of the camera. SliceDrop does not feature an Annotation System like *ScanView* does. *SliceDrop* does allow for custom loading of color tables, whereas *ScanView*'s final implementation relies on predefined colortables. SliceDrop also allows for downloading .nii files of currently viewed data, which *ScanView* is not able to do. Both programs can overlay labelmaps on top of Volume Files. Although not implemented, there is code in *SliceDrop*'s code base that would enable loading and saving to DropBox. So while both programs have different feature sets, *ScanView* is better at comparing multiple files and adding annotations. Furthermore it provides a more interactive feel with its Display Layer Management system and adjustable Layouts.

Compared *Papaya* and *DICOM Web Viewer*, *ScanView*'s feature list looks even more favorable. Both programs are fairly light-weight and contain no 3D Views.

Feature	ScanView	SliceDrop	DICOM Web Viewer	Papaya
Loading NII file	Yes	Yes	No	Yes
Loading Other file formats	No	Yes	Yes*	Yes*
Loading multiple files in one session	Yes	No	No	Yes
Error handling for loading files	Yes	No	Yes	Yes
Orthogonal Views	Yes	Yes	Yes	Yes
3D Perspective View	Yes	Yes	No	No
Volumetric rendering	Yes	Yes	No	No
Volumetric and 2D rendering overlaid	No	Yes	No	No
Comparing several files to each other	Yes	No	No	No
Annotations	Yes	No	No	No
Loading of custom colortables	No	Yes	No	No
Loading predefined colortables	Yes	No	No	Yes
Saving of Data	No	Yes	No	No
Online File Sharing	No	No**	No	No
Adjustable Layouts	Yes	No	No	No
Segmentation Tools	No	No	No	No
Animation along Axis	Yes	No	No	No

*Not fully tested **Not currently available but being worked on

4.1.3 Comparing with Desktop-based programs

Naturally due to the limited time frame of this project, *ScanView* can not compete in terms of feature richness with advanced desktop-based applications such as *3DSlicer* or *ITKSnap*, some of which have had many years head start and various programmers working on it. These applications generally feature all the primary requirements (viewing several files, navigation, 2D and 3D views) as well as secondary requirements (segmentation tools, saving of data). So comparing *ScanView* to *ITKSnap* or *3DSlicer* will not be favorable for the former. However comparing it against smaller applications like *Imview* or *MIview* shows that it matches functionality and features reasonably well and even manages to add new ones. Given sufficient time, it should however be possible to reach a similar level of feature completeness of the more feature rich desktop applications, as from a technical standpoint it seems feasible.

4.2 Reviewing of Use Cases

Considering the use cases that were stated at the beginning of this report:

- A user loads two medical images files ... for image comparison and compositing.
- A user wants to create a custom label map by highlighting a certain section of a loaded scan...
- The user wants to create annotations for a file... (and) can save out this files as a custom data file.
- A user wants to communicate with another user who is not present in the same location...

the first and the third have been implemented in this project. The comparison of two medical image files is possible by loading them into two separate Display Layers and Buffers, and then setting the opacity slider and switching or swiping between the Buffers. Secondly, the Annotations feature covers the third use case by enabling the user to create custom Annotations, edit, label, color and to save them out in *JSON* format. As was pointed out by the project supervisor, this could be used for example for annotating a large set of volume files for a specific feature or organ. This data set could then be used to by machine learning algorithms for automatic detection of similar graphical features.

The second and fourth use case have unfortunately not been implemented, but as they both fell into the secondary requirement category, this is not surprising. Implementing the primary

requirements simply took too much time. However at least the case of creating custom label maps is quite within reach with maybe a month of more development time, since it some of the basic methods such as 2D screen to 3D volume (and vice versa) point conversion methods are already implemented.

4.3 Protocol testing across different platforms

In order to judge whether the software fulfills its purpose of running on a variety of operating systems and web browsers, a number of protocols were tested across a range of computer setups. It was tested on the four most commonly used web browsers and on *Windows 7*, *OSX Mavericks* and *Linux Ubuntu*. Additionally a test was run for the Apple I-Pad. The protocols were designed to mimic what a typical user would do with the software as well as to get a comprehensive coverage of all the features.

4.3.1 Protocols

Protocol 1 - Basic Viewing of Volume File

- Create a Display Layer
- Load a medical image data file (.nii)
- Test navigation - pan/zoom/rotate/traverse
- Test window levels
- Test threshold
- Test changing color lookup
- Test the different layouts
- Test switching Camera Views

Protocol 2 - Creation of Annotation

- Create a Display Layer
- Load a medical image data file (.nii)
- Create a new annotation
- Change annotation label
- Change annotation color
- Change annotation vertex points

Protocol 3 - Loading and Saving of Annotations

- Create Display Layer
- Load a medical image data file (.nii)
- Load/Import annotation *JSON* file
- Change annotation color
- Change annotation vertex points
- Save out annotation

Protocol 4 - Labelmaps

- Create Display Layer
- Load a medical image data file (.nii)
- Load a labelmap file
- Set opacity of labelmap file
- Change color lookup for labelmap

Protocol 5 - Display Layer Management

- Create Display Layer
- Load a medical image data file (.nii)
- Create a second Display Layer
- Load a second medical image data file (.nii)
- Toggle between buffers
- Test opacity slider
- Test horizontal swipe
- Delete First Display Layer
- Test basic viewing
- Delete Second Display Layer

4.3.2 Testing Environments

OSX Mavericks

For *OSX* the website was tested on the latest *OSX* Mavericks Operating System, installed on a Mac Book Pro. All protocols were run successfully for both *Chrome* and *Firefox*.

Hardware:
- Model Name: MacBook Pro
- Processor Name: Intel Core i7
- Processor Speed: 2.3 GHz
- Number of Processors: 1
- Total Number of Cores: 4
- L2 Cache (per Core): 256 KB
- L3 Cache: 6 MB
- Memory: 16 GB

Operating System:
- OS X 10.9.4

Browsers Tested:
- Chrome Version 36.0.1985.143
- Firefox Version 31.0
- Internet Explorer - Not available
- Safari Version 7.0.6 (9537.78.2)

Windows 7

For Windows the website was tested on *Windows 7*, running on a virtual machine on a Mac Book Pro with the same specification as the previous section.

Hardware:
- Running via VMware Fusion on same MacBook Pro as used for testing OSX Mavericks

Operating System:
- Windows 7 Professional N Service Pack 1

Browsers Tested:
- Chrome Version 36.0.1985.143
- Firefox Version 31.0
- Internet Explorer 8.0.7601.17514

- Safari Version 5.1.7

Linux Ubuntu

For *Linux* the website was tested on *Ubuntu*, running on a virtual machine on a Mac Book Pro with the same specification as the previous section.

Hardware:
- Running via VMware Fusion on MacBook Pro as used for testing OSX Mavericks and Windows 7

Operating System:
- Ubuntu 14.04 LTS

Browsers Tested:
- Chrome Version 36.0.1985.143
- Firefox Version 28.0
- Internet Explorer - Not available
- Safari Version - Not available

iOS

For *iOS* the website was tested on an Ipad Air. It was fairly clear that this would not work well, since *iOS* is a closed system that does not allow a free file structure as required by this software. However it was still checked to get a wider coverage of platforms.

Hardware:
- Ipad Air

Operating System:
- iOS 7.1.2

Browsers Tested:
- Mobile Safari Version 9537.53

4.3.3 Protocol Results

OSX Mavericks

All protocols were run successfully for both *Chrome* and *Firefox*. When using *Safari*, they were issues with loading Annotations and Navigation controls. This should be fixable by simply investing more time into the project.

Protocol	Chrome	Firefox	Internet Explorer	Safari
1	Yes	Yes	-	No
2	Yes	Yes	-	No
3	Yes	Yes	-	No
4	Yes	Yes	-	No
5	Yes	Yes	-	No

Windows 7

All protocols were run successfully for *Chrome*. The file loading failed for *Firefox* and therefore all subsequent tests failed as well. However when testing on a machine in Imperial College, *ScanView*

did work perfectly on *Firefox*, so more tests will have to be performed to analyse the source of this discrepancy. There were lots of issues with *Internet Explorer*, which was expected since it is renowned for being difficult to code for and not following conventions of the other browsers. *Safari* was not tested on *Windows 7*, since a stable version could not be installed.

Protocol	Chrome	Firefox	Internet Explorer	Safari
1	Yes	No	No	-
2	Yes	No	No	-
3	Yes	No	No	-
4	Yes	No	No	-
5	Yes	No	No	-

Linux Ubuntu

On *Ubuntu*, both *Chrome* and *Firefox* performed well. *Chrome* worked perfectly as intended. With *Firefox*, some layout issues were notices, for example that the Labelmap Tab was beneath the Annotation tab and not next to it. *Safari* and *Internet Explorer* were unfortunately not available for testing.

Protocol	Chrome	Firefox	Internet Explorer	Safari
1	Yes	Yes	-	-
2	Yes	Yes	-	-
3	Yes	Yes	-	-
4	Yes	Yes	-	-
5	Yes	Yes	-	-

iOS

As expected, most functionality did not work on the Ipad. The display layer management worked fine, however the user is not able to load a file, since the default file opening only allows opening photos. However, it would be possible to circumvent this by implementing a solution that uses a file loading app, such as *Dropbox*.

Protocol	Chrome	Firefox	Internet Explorer	Safari
1	-	-	-	No
2	-	-	-	No
3	-	-	-	No
4	-	-	-	No
5	-	-	-	No

Looking at the results from all browser tests, *ScanView* seems to be stable for *Chrome* on all three platforms. This is encouraging, since most time was spent developing for this platform. *Firefox* support comes in second place, as it works fine on both *OSX* and *Ubuntu*, but not on some *Windows 7* computers. For *Internet Explorer* and *Safari*, *ScanView* fails to supply even basic functionality of file loading, and more time would have to be spent getting it working for these two browsers.

4.4 CPU performance

It has to be noted that the CPU usage is higher in *ScanView* than the original *XTK* apps. This is large part due to the design choice of copying the renders from the *XTK* renderers as discussed in the implementation. This accounts for the majority of the computational overhead. This tradeoff was consciously made to add extra functionality to the Viewer Panels.

4.5 Usage and User Feedback

4.5.1 Google Analytics

At the time of writing, the following results were collected from *Google Analytics* about the usage of the website.

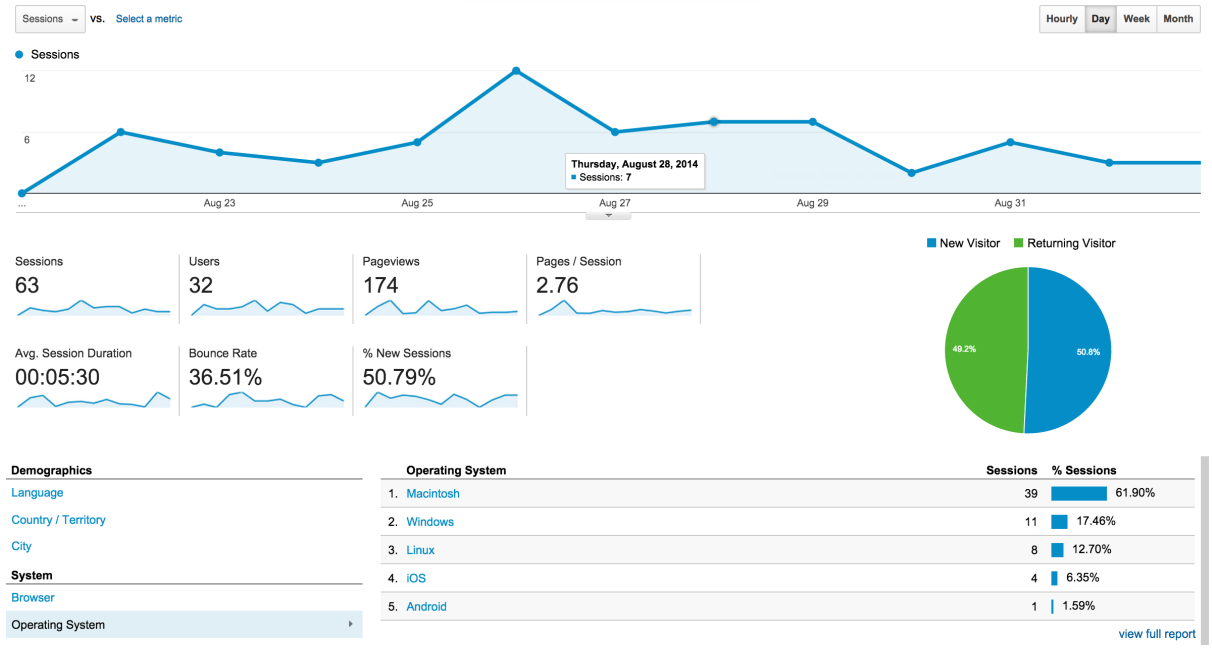


Figure 45: Google Analytics Website Usage Statistics as of September 2nd, 2014

The website has had 32 unique visitors over the 13 days since going live on August 21, 2014. The average session time was 5 minutes and 30 seconds. The majority of users viewed *ScanView* with *Chrome* (74.6%), the recommended browser, on *OSX* (61.9%). Ideally the website would be online for a longer time to get a more complete picture in terms of usage.

4.5.2 Survey Responses and Feedback

Unfortunately only three people answered the survey, but I received two more individual responses with reviews and suggestions by email. For the survey, 2 of the 3 users had used medical imaging software before (such as *ITKSnap*). The users spent 0 to 30 minutes with the website. For judging convenience, the responses were 'extremely convenient', 'very convenient' and 'moderately convenient', which suggests that convenience was judged to be better than average. When asked if the software could be an alternative to desktop software, the answers were 'agree somewhat', 'disagree somewhat' and 'not sure' which is undecisive, and in retrospect deservedly so, as most desktop software reviewed had a lot more features and development time compared to *ScanView*. In terms of criticism, the following were raised (combined with individual email responses):

- High CPU usage on Ubuntu laptop (with Chrome)
- Loading stack of images rather than isotropic volume does not work
- Cannot delete Labelmaps
- Lack of Undo functionality
- Lack of clarity in terms of which orthographical view is which, should be labeled more clearly
- Issue with Opacity and 3D-Views
- Cursor changes to test selection mode during View Panel interaction

- No 3D Views for any Display Layer after second Display Layer

The following additions or changes were suggested from survey and email responses.

- Add support for .nii.gz files (has been added since)
- Add support for other file formats
- Combine "Create Display Layer" and "Open File" actions, as they are implied to go together (has been added since)
- More info could be displayed about the specific files
- Be able to apply colortable selectively to Canvas Views
- Export and save out datasets (in different formats)
- Add a play feature for the 3D view that would rotate a camera around the volume data

As mentioned, it would have been beneficial to run a more comprehensive beta test to get more user feedback for *ScanView*, which unfortunately was not possible in the time of this project (as development took up so much time). Five responses does not quite provide enough data to get a sense of how *ScanView* is perceived by users in general. It would probably help to incentivise survey participation somehow, in order to get more responses. However the answers I did get were very helpful and useful in pointing out deficiencies. I managed to add some fixes to the few criticisms that I received. File loading error handling is much more robust now, with enabled *.nii.gz* support. Some spelling mistakes have also been fixed, as well as some name changes made in order to be less confusing. It is frustrating to receive bug reports of errors that I am not able to reproduce, and could therefore have to do with the specific hardware of the user. At this point it shows that an anonymous survey is not ideal to tackle this kind of problem, and a direct dialogue with users would be preferable. Alternatively to the survey, I could have added a more detailed bug report page, where the user could have specified their browser, version and hardware. Ideally this could be detected automatically like *Google Analytics* already does to a degree.

4.5.3 Outstanding Feedback

I sent a link to the software to the *XTK* developers in order to get their feedback, but at this point in time have not heard back from them. Also two radiologists in London and India have been contacted through private channels, but their feedback is also outstanding at this point.

5 Conclusions

In terms of investigating whether it is possible to create a feature rich browser-based medical image viewer, the course of the project seems to suggest that it is indeed possible given current Web technology. The standards introduced with *HTML5* such as the *Canvas* element and *WebGL* enable richer graphical capabilities, which is required for a medical image viewer. The *XTK* library forms a solid base for adding more features such as in this project.

In terms of creating a functioning piece of software, *ScanView* is successful on *Chrome* and *Firefox*, so is only partially successful in its aim to create a widely accessible tool independent of computer setup. This is due to differing implementation of Web standards between the web browsers. However with more time invested in this, it should be possible to make it work fully across all the most popular web browsers. The problems that currently stand in the way of this are smaller scale issues, and would require small tweaks rather than a whole rebuild.

Furthermore, more time would have been benificial to iron out bugs. Also it would have been great to have a bigger pool of sample data to work from, as there seem to exist certain files that do not open with this software. Given the limited time frame and man power, *ScanView* does not

match the feature richness and performance of high-end software solutions. However this is not a limitation based on the technical possibilities of web browser technology. One issue if ScanView was to be developed further with more features) might be the complexity of scale, as software solutions such as *ITKSnap* require a relatively hefty 30 MBs download. That would be too much to load for a web browser application which should work instantly, so a more complex memory management would be required.

However personally I am satisfied with the progress of this project, in that it mostly achieved its initial goals. It compares favorably to the other web-based medical image viewers in terms of desktop-style interactivity, features count and UI design. It suggests that with a team of dedicated programmers and proper funding, a viable web-browser based medical imaging tool could be created that could rival current desktop based applications. Modern web standards have made this a real possibility and I see no reason why medical image viewers could not exist as web applications along side with *Google Docs*, Autodesk's Webversion of *Autocad* and other complex web- based applications.

6 Future Work

6.1 Limitations and Bugs

At the time of writing, there are a number of known bugs and issues to resolve:

- Annotations are not positioned properly with specific files
- Not all .nii files open, due to issue with *XTK* library
- ColorPicker pop-up for Annotations will open too low in screen if too many Annotation Layers have been created, so that the 'OK' button can not be pressed
- Number input fields for Window and Threshold are too small with bigger numbers
- When panning, object in Viewer Panel does not follow mouse cursor 100%
- High CPU usage has been reported on Linux Ubuntu and Chrome. This is due to a general architecture decision and would take considerable time to redesign item When loading more complex data sets, the responsiveness drops noticeably

6.2 Future Improvements

As has been shown, using a library like *XTK* can be used successfully to write a cross-browser web app for inspecting medical image data. In terms of expanding the software, a number of features could be added which could make the application more useful and user-friendly.

- Add support for other file formats other than .nii
- Re-enabling the loading of custom color tables
- DropBox support could be built into the software to enable file loading, saving and sharing. Additionally this could enable the tool to be used on iOS.
- More work to style the website for different resolution screens.
- Paint tools could be implemented as has been done successfully by BrainBook, to paint custom Labelmap
- Support for 4D data sets could be implemented. This would have be done at a low level in the *XTK* library, but should conceptually not be too hard since it would just be a series of volumes. Appropriate and meaningful user interface additions could be made without much difficulty.
- Smart sliders should be added, so that for Levels and Threshold, the bar can be dragged instead of having to manipulate for ends.
- Interactive splitters could be introduced so that the user has more freedom in defining the size of individual View Panels
- As suggested during the live presentation of this project, alternative input devices such as Leap Motion could be tied into the project.

In this specific case, I would probably chose to rework the *XTK* library more closely to add the features that I have added with *ScanView*, however at a lower level. There exists a performance trade-off based on the decision to have custom Render Canvases copying image data from the *XTK* canvases. This was done to add extra functionality more easily, but it adds extra computational requirements. It should be possible to add this high level functionality into the *XTK* library itself.

References

[1] Preim, B. & Botha, C. (2014) *Visual Computing for Medicine* Second edition. U.S., Morgan Kaufmann

[2] Parisi, T. (2014) *Programming 3D Applications with HTML5 and WebGL*. U.S., O'Reilly Media

[3] Matsuda, K. & Lea, R. (2013) *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL*. U.S., Addison-Wesley

[4] Lindley, C. (2013) *DOM Enlightenment*. U.S., O'Reilly Media

[5] Osmani, A. (2013) *Backbone.js Applications*. U.S., O'Reilly Media

[6] McFarland, D. S. (2009) *CSS the missing manual*. Second Edition. U.S., O'Reilly Media

[7] *Xtk Toolkit* Available from: `https://github.com/xtk/X#readme` Accessed 5th September 2014

[8] *HTML5 Canvas* Available from: `http://en.wikipedia.org/wiki/Canvas_element` Accessed 10th July 2014

[9] *Scalable Vector Graphics* Available from: `http://en.wikipedia.org/wiki/Scalable_Vector_Graphics` Accessed 11th July 2014

[10] *NIfTI-1 Data Format* Available from: `http://NIfTI.nimh.nih.gov/NIfTI-1` Accessed 10th June 2014

[11] *The NIfTI file format* Available from: `http://brainder.org/2012/09/23/the-NIfTI-file-format/` Accessed 9th June 2014

[12] *Neuroimaging in the Browser using the X Toolkit* Available from: `http://www.frontiersin.org/10.3389/conf.fninf.2014.08.00101/event_abstract` Accessed 8th June 2014

[13] Ashkenas, J. *BackboneJs* Available from: `http://backbonejs.org` Accessed 4th September 2014

[14] Ashkenas, J. *UnderscoreJs* Available from: `http://underscorejs.org` Accessed 4th September 2014

[15] *RequireJs* Available from: `http://requirejs.org` Accessed 3rd September 2014

[16] *Twitter-Bootstrap* Available from: `http://getbootstrap.com` Accessed 23rd August 2014

[17] *W3 - Graphics* Available from: `http://www.w3.org/standards/webdesign/graphics` Accessed 17th July 2014

[18] *Brainbook* Available from: `http://users.loni.ucla.edu/~pipeline/pain` Accessed 10th August 2014

[19] *Papaya* Available from: `http://ric.uthscsa.edu/mango/papaya.html` Accessed 4th September 2014

[20] *Weta Digital Purchases Site License Of Nuke* Available from: `http://www.fxguide.com/quicktakes/weta-digital-purchases-site-license-of-nuke` Accessed 30th May 2014

[21] *Industrial Light & Magic Purchases Nuke Site Licence* Available from: `http://www.creativeplanetnetwork.com/the_wire/2009/06/09/industrial-light-magic-ilm-purchases-nuke-site-licence` Accessed 30th May 2014

[22] Alun Evans, Marco Romeo, Arash Bahrehmand, Javi Agenjo, Josep Blat, *3D graphics on the web: A survey* Available from: http://www.sciencedirect.com/science/article/pii/S0097849314000260. Interactive Technologies Group, Universitat Pompeu Fabra, Barcelona, Spain - Accessed 10th June 2014

[23] *Global Apple Iphone Sales* Available from: http://www.statista.com/statistics/263401/global-apple-iphone-sales-since-3rd-quarter-2007/ Accessed 30th August 2014

[24] *July 2014 Market Share* Available from: http://www.w3counter.com/globalstats.php?year=2014&month=7 Accessed 5th September 2014

[25] *Browser Statistics* Available from: http://www.w3schools.com/browsers/browsers_stats.asp Accessed 5th September 2014

[26] *How HTML, CSS and JS work together* Available from: http://webdesignfromscratch.com/html-css/how-html-css-js-work-together/ Accessed 20th August 2014

[27] *Model View Controller* Available from: http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller#mediaviewer/File:MVC-Process.svg Accessed 5th September 2014

[28] *WebGL Water* Available from: http://madebyevan.com/webgl-water/ Accessed 5th September 2014

[29] *ColPick* Available from: http://colpick.com/plugin Accessed 5th September 2014

[30] *22 Experimental WebGL Demo Examples* Available from: http://www.awwwards.com/22-experimental-webgl-demo-examples.html Accessed 5th September 2014

Appendices

Survey Monkey Answers

Q1: Have you worked with medical image data before?

Yes

Q2: If yes, in what capacity have you worked with medical image data before?

Cardiac image analysis.

Q3: Have you worked with medical image viewing software before?

Yes

Q4: If yes, please states which programs you have used!

itkSNAP, rview, SanteSoft.

Q5: How much time did you spend with the ScanView program?

0 - 15 Minutes

Q6: How convenient is the ScanView program to use?

Extremely convenient

Q7: Do you think the ScanView program could be a viable alternative to comparable non-browser-based Medical Imaging Viewing software (eg. Slicer3D)?

Agree somewhat

Q8: Do you have any criticisms of the ScanView program?

Respondent skipped this question

Q9: What features would you like to see added to the ScanView program?

More LookUpTables for the label map.

Figure 46: Survey Monkey Answer 1

Q1: Have you worked with medical image data before?

No

Q2: If yes, in what capacity have you worked with medical image data before?

Respondent skipped this question

Q3: Have you worked with medical image viewing software before?

No

Q4: If yes, please states which programs you have used!

Respondent skipped this question

Q5: How much time did you spend with the ScanView program?

15 - 30 Minutes

Q6: How convenient is the ScanView program to use?

Very convenient

Q7: Do you think the ScanView program could be a viable alternative to comparable non-browser-based Medical Imaging Viewing software (eg. Slicer3D)?

Not sure

Q8: Do you have any criticisms of the ScanView program?

could not get a volume rendering image with the sample data

Q9: What features would you like to see added to the ScanView program?

home button in the 3D view
overlays for rotation and zooming/scaling

Figure 47: Survey Monkey Answer 2

Q1: Have you worked with medical image data before?

Yes

Q2: If yes, in what capacity have you worked with medical image data before?

3rd year PhD student working on fetal MRI

Q3: Have you worked with medical image viewing software before?

Yes

Q4: If yes, please states which programs you have used!

ITKSnap, rview from IRTK, ImageJ, ParaView (VTK-ITK)

Q5: How much time did you spend with the ScanView program?

0 - 15 Minutes

Q6: How convenient is the ScanView program to use?

Moderately convenient

Q7: Do you think the ScanView program could be a viable alternative to comparable non-browser-based Medical Imaging Viewing software (eg. Slicer3D)?

Disagree somewhat

Q8: Do you have any criticisms of the ScanView program?

- high CPU usage on my laptop (Ubuntu and Chrome)
- typo: "Indeces" instead of "Indices"
- what are the main differences with http://slicedrop.com/ ?
- the images are not loaded using their native coordinate system: if you have a stack of slices instead of an isotropic volume, it does not look nice at all

Q9: What features would you like to see added to the ScanView program?

support for *.nii.gz would be nice

Figure 48: Survey Monkey Answer 3

Feedback from Email

Dear David,

great work! Here my observations (unsorted)
- it's strange that the cursor changes to text selection mode during interaction
- no 3D planes for any datasets after third layer (layer one and two keep the 3D view though)
- is it possible to load also different formats?
- is it possible to export and save datasets (also in different formats)
- right click gives strange context menu
- I like the colour table feature
- transparent layers work only between the first two layers
- is it possible to show the cross-hair in the plane views? (should be possible to deactivate)
- I like the play function -- is there an option to export the sequence as gif or avi? The 3D-view could also have a play button, making the camera rotate in the current plane around the centre point (orbital camera)

Figure 49: Feedback on ScanView received via Email