Progressive Volume Rendering using WebGL and HTML5

by

Lisa Tumbleton, B.Sc. Computer Science

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2011

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Lisa Tumbleton

August 31, 2011

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Lisa Tumbleton

August 31, 2011

Acknowledgments

Firstly I would like to thank my parents for the support they've given me for the past twenty four years; my dissertation supervisor and course director John Dingliana for his advice and guidance throughout the course; my IET classmates and the alumni from previous years for making the last twelve months a great experience and all my other friends and family for their patience and friendship.

Lisa Tumbleton

University of Dublin, Trinity College
September 2011

Progressive Volume Rendering using WebGL and HTML5

Lisa Tumbleton
University of Dublin, Trinity College, 2011

Supervisor: John Dingliana

Volume rendering has been a research topic of great academic interest since it's emergence with the introduction of medical scanners such as Computed tomography (CT) in the 1970s. The data gathered allows medical practioners to visualise both the internal and external structure of their patients. The use of volume data has been adopted since in other fields to assist with the simulation of materials and fluids. Previously visualising this data was only possible using powerful computers with third party three dimensional (3D) rendering software installed, consequently limiting the amount of client computers that were capable of running the applications.

This dissertation proposes a method to display volume rendering through a modern web browser using HTML5 and WebGL exclusively. This eliminated the need for additional non-standard plugins or software and consequently made the application

universally accessible to the average computer user.

Current state-of-the-art methods for volume rendering are analysed and one approach is adapted to work in WebGL based on contemporary limits and capabilities of WebGL enabled browsers. Emphasis is placed on giving the user a volume rendering application that is compliant with the customary use of the browser environment. To this end, a method is proposed to load in the volume data progressively so that the application does not appear to freeze and unintentionally give clients the perception that the web page has failed to load. Secondly, this paper will examine the use of hardware accelerated rendering techniques to improve the features and multidimensional cognitive perception of the volume information (particularly cutting slices and realistic gradients) with the use of the OpenGL Shading Language (GLSL).

The outcome will be a volume rendering application available over the internet which is capable of real-time interactive frame rates. Furthermore, the user will have the ability to instantaneously change the visual appearance of the data by manipulating various display properties available through the user interface (UI) on screen.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation will contain six main chapters.
The chapter listing is as follows:

1. Introduction.
2. Background and Related Work.
3. Design.
4. Implementation.
5. Evaluation.
6. Conclusions.

In the introductory chapter, the basic concepts of volume rendering will be explained in detail. The current applications of volume rendering and its potential future uses will also be discussed. A brief overview of WebGL will then be given. This will include a short section on the current state of the internet. Following this, the motivation and goals behind the dissertation will be outlined.

The background and related work chapter will outline various methods for volume rendering techniques detailed in previously published papers. Following this, modern advances in the graphical illustration of the aforementioned volume renderings will be discussed. Lastly, since WebGL is a relatively new technology a small selection of papers speculating over the potential future applications of WebGL shall be analysed.

The chapter covering the design aspects of this dissertation will outline the functionality required by the system. It will detail the features of the application and the approaches that were taken to achieve them.

The fourth chapter will discuss the implementation of the core system. It will describe in detail how the previously designed system was constructed. The architecture of the system will be defined along with the algorithms and technology employed.

Chapter 5 will review the application based on the initial goals and evaluate the end results acquired. The system will be judged under different headings such as performance and visual output.

In the last chapter, conclusions will be drawn about the application. We will discuss and review the proposed system based on the original aims of the dissertation and theorize future additional features or modifications that could be adapted by the application.

1.1 Concepts of Volume Rendering

The term volume rendering was first coined concurrently, in academic papers by authors Drebin, Carpenter and Hanrahan [11] and Levoy[9] in 1988. The term volume rendering has come to mean a method by which the content of a substance and its distribution is manifested in a manner intelligible to the eye without the use of any pre-constructed geometric surfaces that were created to represent the data.

The three dimensional data sets that are employed to represent volumes consist of items called "voxels". A voxel is a *portmanteau* word, meaning that it is a blend of several words to create one new word. In this case, voxel comes from the idea of a volume element or volumetric pixel. As a pixel represents a point on a two dimensional (2D) image, a voxel depicts information at a position in a 3D grid. Each voxel will then contain information about itself. For example it could contain a scalar value to represent the density, or a vector to describe the normal of the volume at that point. The greater the amount of voxels in the grid, the higher the resolution of the volume will be, e.g., where a picture might have 128 x 128 pixels, a volume could have 128 x 128 x 128 voxels.

Voxels first arrived in the early 1970's with the inception of Computed Tomography (CT) or Computed Axial Tomography (CAT) scanners. This technology, which is now

commonplace in the majority of hospitals, produces computer generated images of the internal and external structures of objects by taking x-rays at different locations and converting the results into a 3D dataset.

Over the years, volume rendering has become a very important technology in relation to medicine (for internal visualisation), to science (for the simulation of substances e.g. fluid) and to the entertainment industry (for reproducing cinematic effects such as smoke or water).

1.2 History and development of web browsers

Web browsers are software applications that allow computer users to view web pages hosted over the internet. The first web browser was created in 1991 by Tim Berners-Lee [12]. It was originally called "WorldWideWeb" but was renamed later to "Nexus" to avoid any confusion between the software and the World Wide Web.

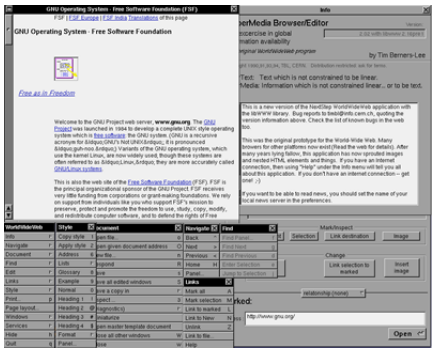

Figure 1.1: Nexus Web Browser (WorldWideWeb browser)

It wasn't until 1993 and the release of one of the first graphical web browsers that a large increase in web use was seen. This browser called Mosaic, was developed by the National Center for Supercomputing Applications (NCSA) at the University of Illinois. Mosaic was the first browser capable of displaying images inline with text, in older browsers images where opened in seperate windows. Marc Andreessen, one of the creators of Mosaic, then started his own company and released the Netscape Navigator browser in 1994, which quickly became the most popular browser at that time with 90% of all web traffic.

Figure 1.2: Netscape Navigator browser 1994

In 1995, Microsoft released the Internet Explorer browser, bundling it with their Windows operating system. Internet Explorer quickly became the dominant web browser. This release marked the start of the "browser war" which remains ongoing.

Other notable releases include Opera released in 1996, Apple's Safari browser released in 2003, Mozillas Firefox in 2004, and most recently Googles Chrome in 2008. Since the infancy of the internet its user base has grown consistently. The current estimation of internet users is 2,095,006,005 which is 30% of the world population [13].

Figure 1.3: Google Chrome browser

As browsers have grown more sophisticated over the years so too has the language to display webpages. HTML which stands for HyperText Markup Language was first

specified in 1990 by Tim Berners-Lee (the aforementioned creator of Mosaic). It has gone through much iteration since and is, as of August 2011, in the development stage of its fifth version, referred to as HTML5. The important feature of HTML5 to note is that this version aims to improve the language while being designed to make it easier to add multimedia and graphical content on the web without having to resort to proprietary plugins or APIs (Application Programming Interface).

The new feature that will be examined in this dissertation is the HTML5 canvas element. This element allows for the use of WebGL inside web pages. WebGL is a software library that allows for the generation of 3D graphics within a browser with WebGL capabilities. It extends upon the Javascript programming language and therefore can be used without the use of plugins. WebGL is an API that is based closely on the OpenGL ES 2.0 standard API. OpenGL is a 3D graphics interface that can be used a wide set of platforms whereas OpenGL ES is a smaller subset of OpenGL specifically created for the use on embedded system such as phones and mobile devices. WebGL is a recent advancement in technology; the specifications for version 1.0 only being released in March 2011.

1.3 Motivation

To date, volume rendering, has been only been accessible to a small subset of computers users. This is mainly due to the non standard software that must be installed to run the rendering applications thereby limiting the number software users to those with technical backgrounds to operate and install the programs.

The aim of this dissertation is to investigate the use of WebGL to make volume rendering accessible to the average user with a modern internet browser. [7] presents a tool for volume rendering in the browser using Java and the Virtual Reality Modelling Language (VRML) however this means that the user must install extra software for it to run. WebGL allows for the rendering of 3D objects in the browser devoid of any additional software making it accessible to all internet users with WebGL enabled browsers.

Currently as of August 2011 there are three browsers available offering WebGL capabilities. These are Google's Chrome browser, Apples Safari browser and Mozillas Firefox browser. It should be noted that recent statistics acquired about the current

state of the web show that the most commonly used browser is Microsofts Internet Explorer which offers no WebGL support. Even more interesting though is the obvious trend which can be seen in statistics taken from over the past few years which show that smaller web browsers are growing in popularity while Internet Explorer is consistently losing its clients, meaning more people are switching over to WebGL enabled browsers every year.

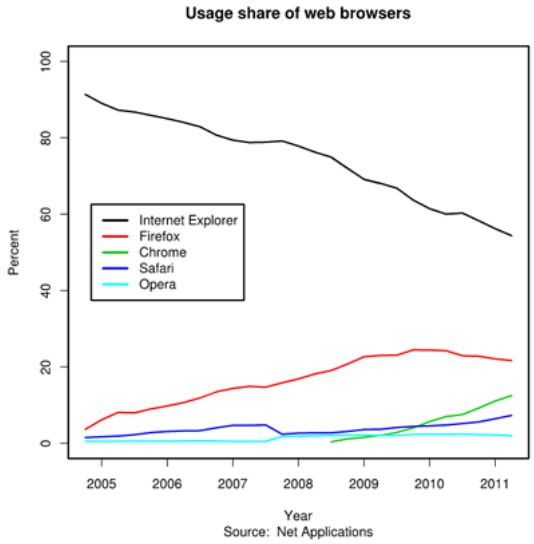

Figure 1.4: Usage of web browsers 2005 - 2011

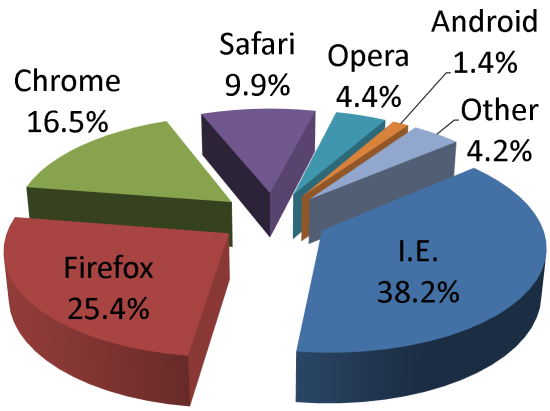

Figure 1.5: Browser Usage from WikiMedia, June 2011

Chapter 2

Background and Related Work

2.1 Volume Rendering

Volume Rendering is the term usually given to describe the visualisation of 3D datasets. Originally these 3D datasets where used to describe items of a scientific nature e.g. magnetic resonance images (MRI), and since their invention finding appropriate ways to visually represent this data has been an important area of research.

Even though modern volume rendering has many different uses, e.g. the simulation of clouds by Harris et al [14] these varying modern day uses face the same problems as the datasets for medical imaging, how to accurately portray a 3D volume within a 2D space. Albeit an accurate visualisation of the 3D information is a more important requirement from the medical imaging stand point as a physician may need to form a diagnosis from the information displayed to them and as such they need to know that said information is trustworthy. A good example case is given in the start of Stytz et al. [15] where it tells of how a patient with intractable seizure activity is admitted to a major hospital for treatment. As the first step to finding out the cause of the ailment, the hospital orders a MRI on the patient and collects 63 image slices of the patients head. By observing the 2D images obtained from the MRI an abnormality cannot be observed. A 3D model of the MRI study however, reveals an irregularity that was not apparent in the cross-sectional MRI views.

There are two main bottlenecks that volumetric rendering faces. These are processing the significant amount of data in each volume (medical MRIs are typically between

2mbs and 35mbs in size) and allowing for the manipulation of the resulting image in real time.

Volume datasets also suffer from added complexity since there is no single accepted file format. Many different types of information storage appear in modern computer graphics, to cater for all these various formats separate data loaders must be implemented. A file format used frequently is Dicom [16] Dicom is the Digital Imaging and Communications in Medicine Standard and was developed for use with medical imaging devices.

Various methods for volume rendering have been developed over the past few decades to display the 3D data to the user through a range of different programming languages. Each method has its own advantages and drawbacks.

In the next, section I shall describe the different techniques used to display volumes on modern computing systems. These techniques can be separated into two different sections; indirect and direct volume rendering.

2.1.1 Indirect Volume Rendering

Indirect Volume Rendering is where the system for displaying the volume data doesn't have access to the original volume information. The system is supplied with a post-processed version of the volumetric data, usually a polygonal model of the data. The 3D model is generated by extracting surface information from the original volume data. Then it is up to the system to render the model on screen to the user. This was one of the earliest forms of volume rendering proposed. As computer hardware improved more methods of direct volume rendering were introduced. Models created from indirect volume rendering are used extensively in modern surgery simulators to create a graphical representation of the locale in which the surgery will take place.

An early technique by Keppel [17] creates contours to be constructed from the data but errors arise in this method when multiple contours are located on the same slice. A later paper [1] builds on this technique while also presenting a general solution to the problem where a surface must be constructed over a set of cross-sectional contours.

Another early approach (similar to the volume rendering used in the modern game Minecraft by Swedish creator Markus Persson) divides the 3D space into cubes of equal sizes by three orthogonal sets of parallel planes e.g. along the x, y and z planes. This

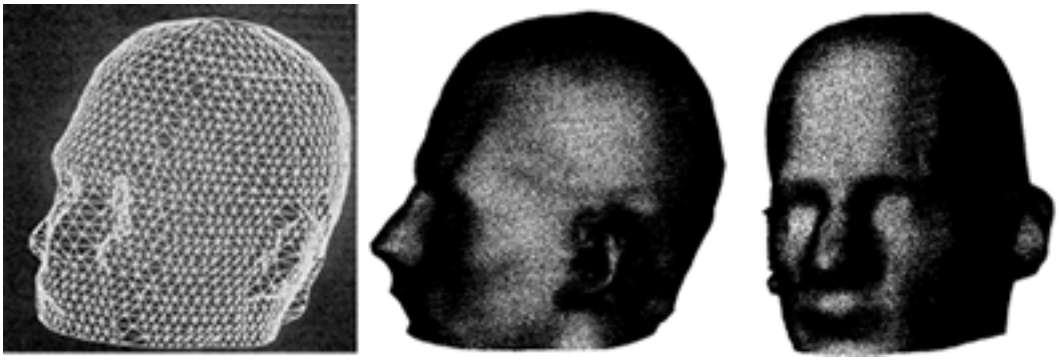

Figure 2.1: Indirect Volume Rendering [1]

approach was developed by Herman [18] the model surfaces are then created from rendering these cubes and the cubes gradient is determined from the neighbouring voxel information.

In 1987 Lorensen et al. proposed a new way to extract the surface geometry called marching cubes [19]. This method creates models of constant density surfaces using a divide and conquer approach to calculate the connectivity of surfaces between slices. It generates each connection between triangles be examining 8 neighbouring voxels from two adjacent slices (4 from each slice). The algorithm then determines the surface of each cube out of a possible 256 types of intersections and once complete moves on to the next cube.

There have been many advances within the area of indirect volume rendering, a researcher named Marc Levoy of North Carolina University has made many similar contributions to volume rendering including using points to display volume data [20] and extracting geometric primitives from volume data [9].

A more recent contribution by Kobbelt et al. [21] extends upon Lorensens marching cube algorithms and an enhanced distance field representation to extract models of finer detail. In their paper, they present a well known fandisk dataset that has been recreated to a approximation error of below 0.25

2.2 Direct Volume Rendering

Direct volume rendering differs from indirect volume rendering in that when it renders the volumetric data, it does so by retrieving rendering data straight from the 3D

dataset and doesn't involve the use of precompiled polygonal geometric structures. It is the most widely used form of volume rendering since it allows for various additional elements including animation and improved rendering of voxels. I shall discuss the main varieties of direct volume rendering in the rest of this section.

2.2.1 Splat Based Volume Rendering

Splatting is a well-known technique for volume rendering. It, like the shear-warp method (described later), trades accuracy for increased speed. It was first proposed in 1990 by Westover [22]. It is performed using a forward mapping rendering algorithm to display volumes data in a 3D grid regardless of the volumes spacing along the x, y and z planes. An improvement on this technique was proposed in 1991 by Laur and Hanrahan [2] where a progressive refinement algorithm for volume rendering using a pyramid like volume representation is fitted with a octree. The octree is then drawn using a set of splats, or footprints. The splats themselves are small simple polygons drawn using Gouraurd shading to increase efficiency during render time.

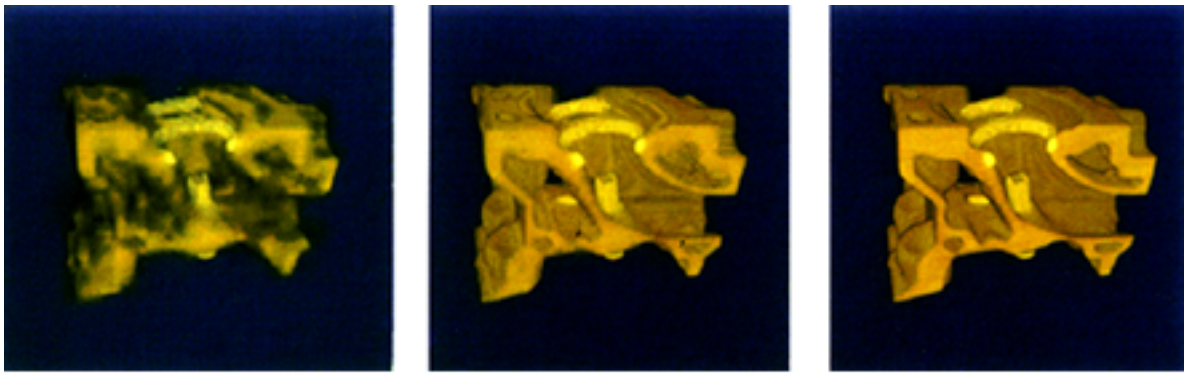

Figure 2.2: Engine block, Splat Based Volume Rendering [2]

2.2.2 Texture Slice Based Volume Rendering

2D Texture Stacking Volume Rendering

2D texture stacking is a volume rendering method that utilises modern graphics hardware and transparency rendering to illustrate the volume. It does so by mapping values from the volume information on to textures that are bound to polygonal planes. Three

stacks are generated to represent each axis. Each stack is made up of various texture mapped planes. To generate correct transparency values the planes are sorted and then rendered in back to front order. This methods major advantage is that it's fast and efficient but can have some inaccuracies when viewing the image at extreme angles. An example of it in use can be seen in [7].

Shear-Warp Volume Rendering

Shear-Warp Volume Rendering was first proposed by Cameron and Undril [21] and then popularised by Lacroute et al. [3] in 1994. The algorithm in the paper describes a method that is based on a factorization of the viewing matrix into a 3D shear parallel to the slices of the volume data, a projection to form a distorted intermediate image and a 2D wrap to produce the final image. The algorithm is relatively fast but suffers less accurate sampling and a potentially worse image quality. This method can be optimized using run length encoding, which is where sequences of data in which the same value occurs consecutively can be stored as a single data value and count.

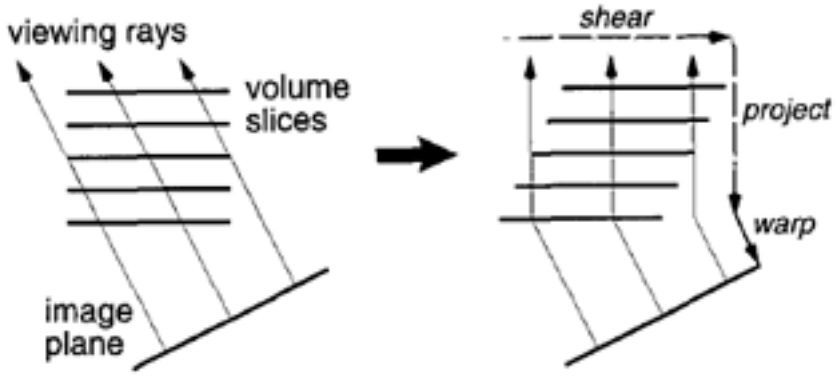

Figure 2.3: Shear-Warp Parallel projection [3]

3D Texture Volume Rendering

3D Texture based volume visualisation is a rendering method which utilises graphics hardware that supports 3D textures. With the advent of very fast texture mapping hardware in modern computers, volume rendering techniques which implement texture

mapping have become an important area of research. One well known method was introduced by Cabral, Cam and Foran in 1995 [4]. It allows for the rendering of 2D slices of 3D volume data and real time interaction with that data. The general idea is that the volume data is interpreted as a 3D texture and that the 3D texture map is understood as the trilinear interpolation of the volume dataset at a point within its boundaries. The 2D planes which lie parallel to the viewing plane are then textured by trilinearly interpolating across the 3D texture within the volume.

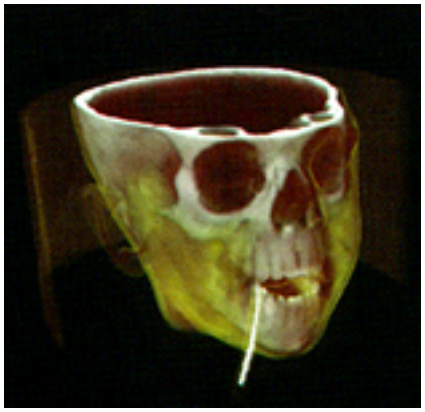

Figure 2.4: 3D Texture Volume Rendering of 512 X 512 X 64 skull [4]

Different methods to speed up the 3D texture mapping process have been developed since the methods inception. One such example can be found in a paper by Westermann and Ert [5], in this paper they introduce the concept of clipping geometries by means of stencil buffer operations. The paper also demonstrates a way to map volume data to spherical domains.

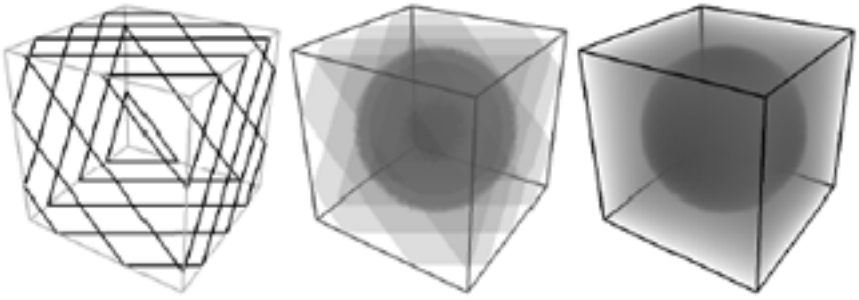

Figure 2.5: 3D Texture Slicing Volume Rendering [5]

2.2.3 Ray Casting Volume Rendering

3D texture based volume rendering mentioned above has some disadvantages when it comes to representing large datasets of volume information. The overhead can be quite large since computations must be made on all voxels in the space even though a significant number of these voxels might not contribute to the final image. For each fragment a ray of sight is cast through the volume sampling the colour at each slice until the ray leaves the volume. It is an image order algorithm which calculates the colour of a 2D pixel from a 3D volume. The implementation can be quite costly however and because of this ray casting in real time simply wasn't possible over a decade ago.

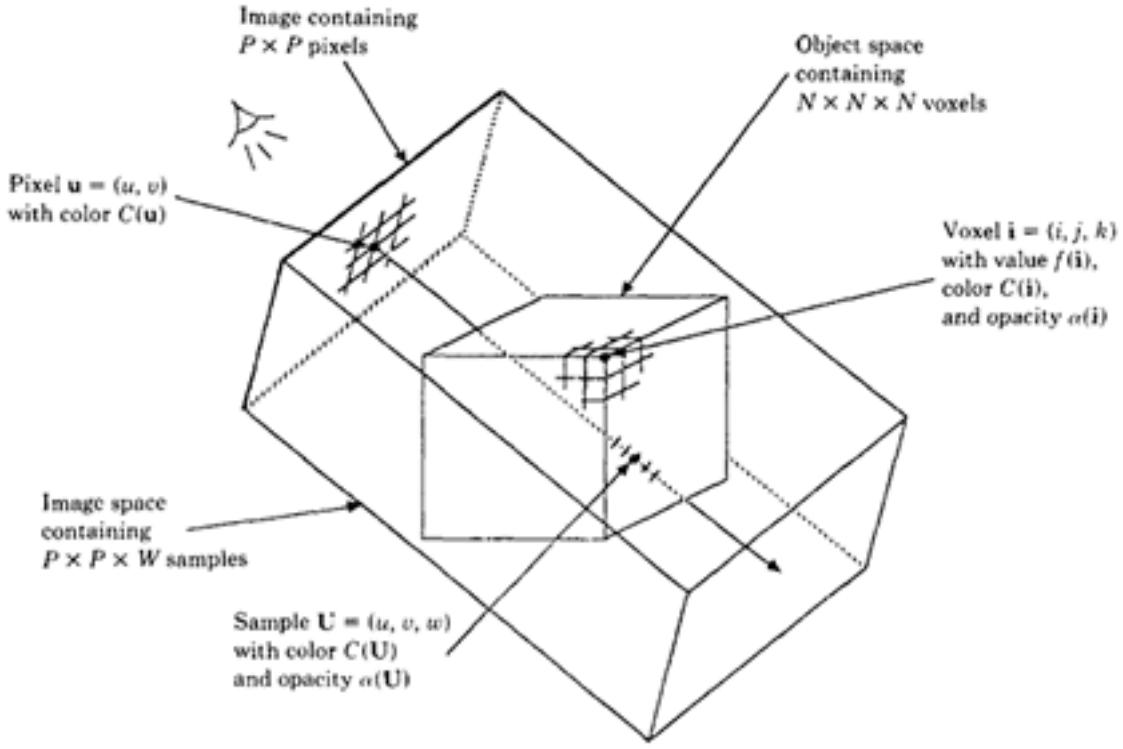

Figure 2.6: Ray Casting Volume Rendering [6]

With the advancement in the processing abilities of GPUs, ray casting in real time became a reality and in 2003 Kruger and Westermann [23] released a paper describing how to implement it on the GPU.

Ray casting is made more efficient by several optimisation methods. Early ray

termination is a optimisation technique commonly used and was first described by Whitted in[24]. This optimisation is based on the idea that once a ray samples information from an opaque object, or has passed through enough semi-transparent objects that the opacity of the ray stabilises, then the ray doesn't need to continue any further into the volume and can be terminated.

Recent important advances in the optimisation of ray casting volume rendering methods can be seen in two papers [25] and [26]. In the GigaVoxels paper [26] proposed by Crassin et al. they detail an approach to rendering large volumetric datasets using an adaptive data representation depending on the current view and occlusion information alongside a efficient ray-casting rendering algorithm. In the second paper, on efficient sparse voxel octrees [25] Laine and Karras propose a new compact data structure for storing voxels and a efficient algorithm for performing ray casts using the new voxel structure.

2.3 Web GL: History and Technologies

Up until recently, content on the internet has usually been a generally 2D experience. It was invented about twenty years ago to connect different nodes using hypertext (which would be standardised as HTML later on) to share reports and information from databases. 3D experiences have been achieved in online games and applications since but this is usually through the utilisation of 3D collaboration programmes that can be used as plugins in browsers. As such, the use of 3D in web browsers is usually confined to specific, non-mainstream uses.

2.3.1 The advancement of browsers

The first browsers were only capable of rendering plain text in HTML form from the internet. It wasn't until 1993 when the US National Center for Supercomputing Applications released the first browser capable of displaying images and text called Mosaic [27]. Javascript is a type of browser scripting language that was invented to allow the user to interact with the content feature on the pages.

Features and capabilities of browsers have been continuously growing and since then several attempts of bringing 3D objects to internet browsers have been proposed

including VRML (Virtual Reality Markup Language) in 1994, X3D a xml based file format for representing 3D graphics developed by the Web3D consortium in 1997 and universal 3D technology proposed by the 3D Industry Forum in 2003 which works in most browsers via a plugin.

Using plugins in browsers have many disadvantages. They are only supported on a limited number of platforms and are heavily dependent on the company that created them. Plugins can easily crash bringing the internet browser to a halt or have security vulnerabilities.

2.3.2 HTML 5

HTML 5 is the new internet standard that is being currently implemented in the latest versions of modern browsers such as Googles Chrome, Mozillas Firefox and Apples Safari browser. One of the main goals or HTML5 is to minimise the need for plugins to display multimedia content or ideally, remove the need for them completely. HTML5 has the ability to cater for video, sound, 2D and 3D graphics amongst other improvements. One of the most important new features HTML5 introduces is the Canvas type element[28]. Developers can use the canvas element in regular HTML pages to implement 2D graphics or 3D graphics without the use of plugins.

2.3.3 Web GL

Web GL is a lightweight software library that uses Javascript to implement OpenGL. ES 2.0 commands[29]. OpenGL ES is a subset of Open GL that has been designed for embedded devices such as smart phones and PDAs. This allows for the 3D information drawn in the canvas element to be compatible not only with browsers on desktops but on small mobile devices too. Open GL and Open GL ES are published by the Khronos Group which is consortium consisting of companies that deal with graphical hardware and software such as Nvidia, Sony and Intel.

Even though Web GL is a massive improvement on what has been available through the browsers before it does have some disadvantages including resource loading limits, security limitations and event handling limitations, more information on these topics are presented by Borsos [30]. Another disadvantage is that since Web GL is based upon Open GL ES it does not include 3D textures (3D textures are only available in

Open GL ES through an extension). This means that the 3D texture volume rendering technique listed in section 2.2.1 cannot be used with Web GL to display volume data.

Chapter 3

Design

The aim of this system is to produce a volume rendering application that is usable through a WebGL enabled browser at real time rates without any additional third party software. The main inputs of the software will be the volume data to be drawn, textures representing various 1D transfer functions and rendering variables modifiable by the user.

3.1 Requirements

The application is required to display to the user the rendered image of the volume data and a graphical user interface (GUI) for the user to interact with the volume through. Since this software will be run through the browser it must be implemented in such a way to avoid any pages freezing which can in turn cause the browser and all other open tabs to freeze.

3.2 System Over View

The system can be broken down into various elements. The following sections will describe each of these elements in greater detail. The core parts of the application are the data loader and the volume renderer subsystems. The loading subsystem section is responsible for the progressive transfer of the volume data from the server to the application on the local client machine. The volume rendering system will use

a variation of a technique described in the state of the art that has been optimised to work for WebGL as well as utilising GLSL pixel shaders available through WebGL to improve the visual illustration of the data.

3.3 Data Loading Technique

Finding a quick and reliable way of loading in the data for volume rendering is a very important aspect of any volume rendering application. The importance of data loading in this system is even greater emphasised since the clients browser needs to retrieve the data from the server and download it to the local machine. The speed of the download is also governed by the speed of the clients' internet connection; a low download bandwidth will guarantee the user a slower download.

3.3.1 The Problem

The biggest obstacle which we're faced with is based around how browsers and javascript work in general. Up until very recently it was impossible to have multi-threading within websites. Prior to 2009, Javascript could only emulate multi-threading using techniques like the asynchronous calling of methods.

In a single threaded program, if the thread spends too long trying to complete a certain task the application can appear to freeze. When this happens in Javascript, it means the web page that's being displayed is no longer responsive. Since interactivity is a key element to browsing the internet, the majority of browsers have a timeout limit when pages become unresponsive. Once the timeout limit has been reached the browsers will display an unresponsive script warning and give the user the option of "killing" or stopping the script. The main advantage to this feature is to give the user the option to stop badly written code, for example an infinite loop written in Javascript will mean the browser freezes and the page never loads completely.

The time out limit is different for each browser since its set by the developer. With Firefox the default value is 10 seconds [31]. This presents an obstacle that must be overcome since ideally the user of the application must be able to interact with the web page while it's loading without receiving warning messages from the browser itself about the validity of the code.

3.3.2 Solution

In order to solve the problem of the application freezing during load, our approach utilises a new browser element called web workers. Web workers allow the application to run Javascript scripts in the background of the system. These threads run independently from the main thread and therefore do not interfere with the user interface of the application.

In our application, a web worker will be spawned to load in the volume data files from the server while the main thread deals with the user interaction. This means that the user will be able to modify the volume data in real time as the files are loaded in and that the browser will never inform the user that the page has become unresponsive.

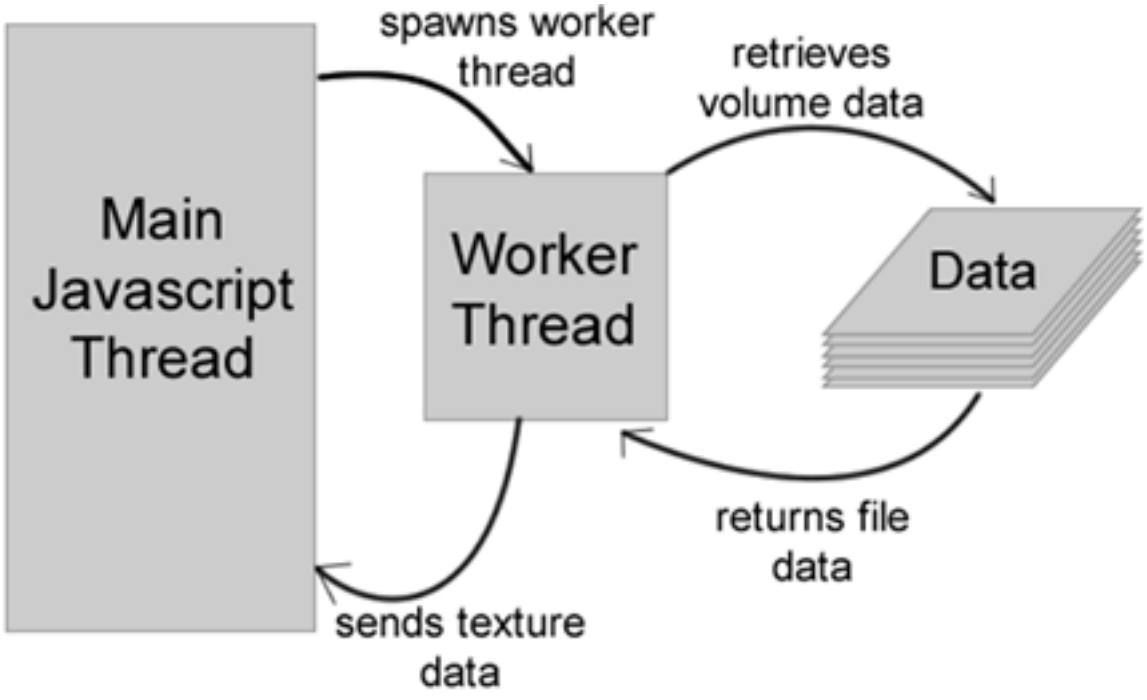

Figure 3.1: Web Worker Data Loader

Currently not all web browsers offer support for web workers, fortunately though all the browsers which support WebGL also support the web worker. Currently as of August 2011, the current list of browsers that support web workers are Chrome, Safari, Firefox, Opera and the Android OS. Although Safari for iOS 4 (iPhone 4 OS) does not

support them, they will be supported by iOS 5.

3.4 Volume Renderer

Choosing the right technique for rendering the volume is a very important task. It must be an accurate and fast system which is compatible with the current limits of WebGL. As mentioned in the state of the art section 2.3.3, one of the limits of WebGL is that it does not support 3D textures. This automatically means rendering the volume using 3D textures isn't possible.

Another approach that was looked at was rendering using ray casting, but since ray casting on the CPU is extremely costly and ray casting on the GPU is only possible with the support of 3D volumes, this method was abandoned.

3.4.1 Our Approach

Out of the remaining choices for rendering the volume, we chose to use the 2D texture stack approach. Splat based volume rendering can be less accurate and the shear warp algorithm is known to cause artefacts when the viewing angle is near 45 degrees relative to the slices of the data. This application by Engel and Ertl [7] using Java and the Virtual Reality Modelling Language (VRML) uses a similar rendering approach.

The volume datasets used in our application are taken from the Stanford volume data archive [32]. The sets from this repository are regularly used in similar papers as test data. The datasets are stored in such a way that there's a separate file for each slice perpendicular to the slice direction. Each file contains no header and is formatted using 16 bit integers in Mac byte ordering.

The web worker from the previous section loads the files concerned with the dataset being displayed. Every time the worker receives a data file, it parses it from 16 bit integers into an array of floats which are then sent to the main thread. Since the data is stored in Mac byte ordering, it's parsed as big endian. Endianness refers to how the bytes are ordered, little endian files store the least significant byte first whereas in big endian files, the most significant byte is stored first.

When the main thread receives the array of data values from the worker, it adds the values to the 3D volume data array and creates a texture using this information.

These textures are then bound to the corresponding polygonal planes and used as texture maps. Once all the files are loaded in and their corresponding plane textures are generated, the web worker than creates arrays containing the isosurface information of the slices from the other two perpendicular axes. These arrays are then sent to the main thread in a similar manner. The application then takes the information from these arrays and generates the textures for their corresponding planes.

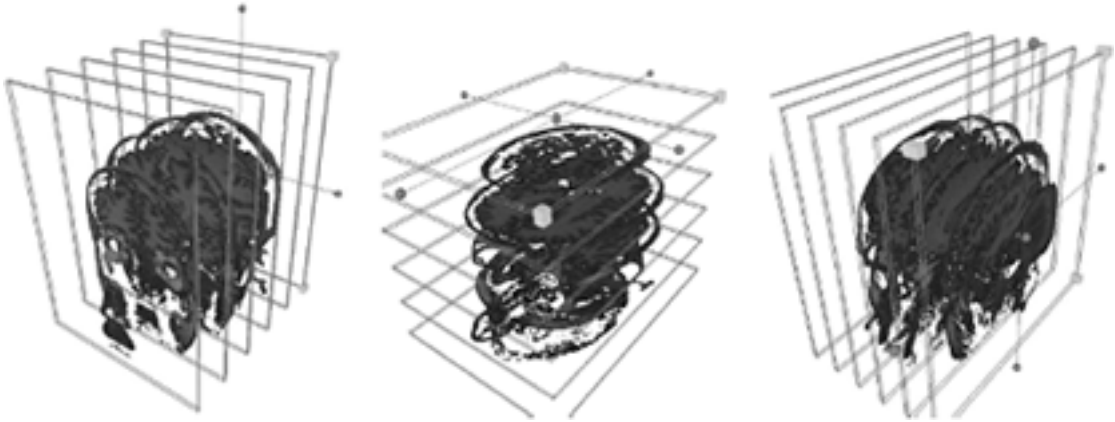

Figure 3.2: 2D texture stacks taken from [7]

The application then chooses which of the three axes stacks to render based on which ones orientation is closest to that of the viewing screen. This means that the set of stacks being drawn will never be more than 45 degrees away from the orientation of the screen. At draw time each texture mapped primitive from the stack is then rendered on screen in back to front order using appropriate alpha blending parameters to correctly display the transparency of the textures.

When the volume rotates in world space, the stack is replaced according to the new relative screen orientation.

3.5 Enhancements

This section will look at the design of various visual enhancements that we've implemented in the application. These extra features were chosen to enhance the usefulness of the application while keeping computational costs low.

3.5.1 Volume Clipping

Cutting planes also known as slice planes are a very helpful feature when it comes to visualising a volume. They allow the user to clip sections of the volume they don't want visualised. Three different cutting planes are implemented in the application which can be moved using HTML5 input sliders in the UI on the web page. Each slice plane lies parallel to one of the three axes, either x, y and z. When the user modifies the position of these planes the application receives notification and updates the display.

When plane objects are created they are each given a position attribute which stores the location of the center of the polygonal plane. When using a clipping plane that is parallel to the slices currently being rendered, slices that are above that position are considered to be hidden by the slice plane and aren't rendered. Slices that are perpendicular to the cutting plane are displayed correctly by passing two separate variables to the GLSL shader.

The GLSL shaders used to render the volume data store these two variables as uniform floats. The two variables correspond to the position of the two clipping planes positioned on axes perpendicular to the current slice axis. The two positions are scaled down to correspond to values from 0 to 1. This is done to correspond with texture mapping coordinates which are always valued from 0 to 1. When each slice is rendered, the fragment shader (or pixel shader) compares the texture position of that fragment to the position of the clipping plane. If the texture coordinate is less than the clipping plane position variable then the opacity of that fragment is set to 0, giving the user the impression that fragment has been cut off.

This method allows the application to have fast volume clipping from multiple cut planes at the same time since the majority of the work is computed on the GPU in the GLSL shader.

3.5.2 1D Transfer Function

Transfer functions are a key element for illustrating the separate materials and substances that make up the volume being displayed. It performs the essential task of surface classification, allowing the user to view the structure of large objects where all the voxels contain similar values. For example, a transfer function can be created to clearly illustrate separate bone and muscle material in the human body. It allows

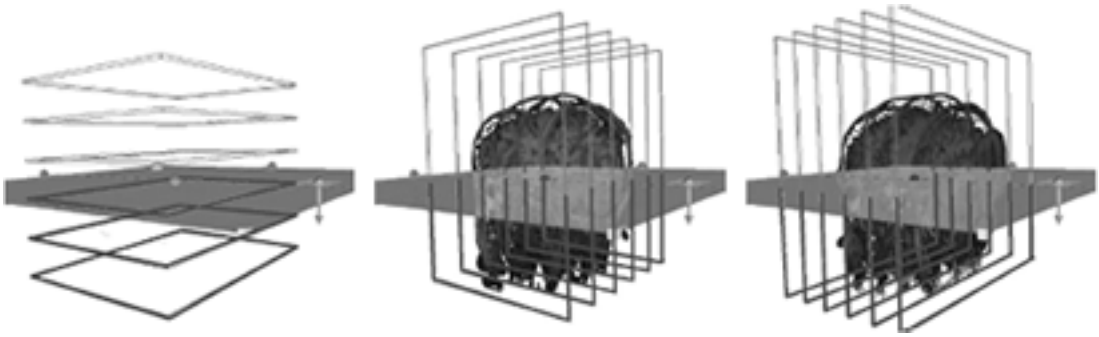

Figure 3.3: 2D Texture Stacking Volume Clipping from [7]

for the mapping of colours to corresponding iso values extracted from the isosurface. A vast amount of research has gone into the automatic generation of transfer functions and also into developing intuitive user interface widgets to create and manipulate transfer functions but these are considered out of the scope of this dissertation.

This application presents the user with three previously made transfer functions to choose from using the UI on screen. Each transfer function is a texture of 256 pixels wide that is passed into the pixel shader through a texture sampler uniform variable and an associated texture channel. The texture has a width of 256 to account for every variant of the isosurface density value, since the isosurface values are 8 bit numbers and therefore restricted to values between one and 256. The transfer textures are generated from .png files containing red, green, blue and alpha channels.

When it comes to the transfer texture, it was decided it was best not to make any all-or-none decisions about which material is present at that isovalue so gradients are used when moving from one classifier colour to another. As stated in [33], having exact threshold limits can introduce unwanted artefacts in the final image so our approach avoids this.

3.5.3 Phong Lighting through Iso Surface Extraction

The Phong lighting model was developed by Bui Tuong Phong at the University of Utah as part of his dissertation that was published in 1973 [34]. The Phong shading model involves the generation of the objects diffuse color using the light direction vector and the normal vector of the fragment to be shaded. Once the GLSL shader has these two vectors, computing Phong shading is a relatively easy process.

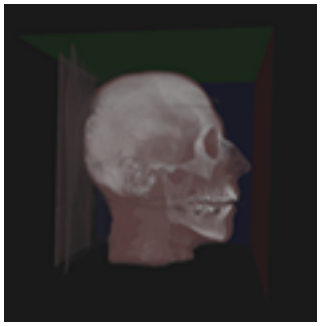

Figure 3.4: Example of 1D transfer function used in application (pink for flesh, white for bone)

$$fragColor = (l \cdot n) * textureColor \tag{3.1}$$

Equation 3.1 shows the basic equation for calculating the diffuse color of a fragment where fragColor is the final output color of that fragment, l stands for the normalised light vector, n describes the normalised normal vector at that location and textureColor is the corresponding fragments color retrieved from the texture map.

Since the volume dataset only supplies the application with isovalues for each voxels density, extra computations must be completed to estimate the normal vector at each voxel. In [9] Levoy desribes a equation to calculate each voxels normal which will be implemented in our application. This equation takes the values from the current voxels surrounding six neighbours and examines the difference of the values between them i.e. the gradient of the change.

Figure 3.6 shows the equation created by Levoy [9] to calculate the normal vector. Where $\bigtriangledown f(x_i)$ is the approximated normal vector.

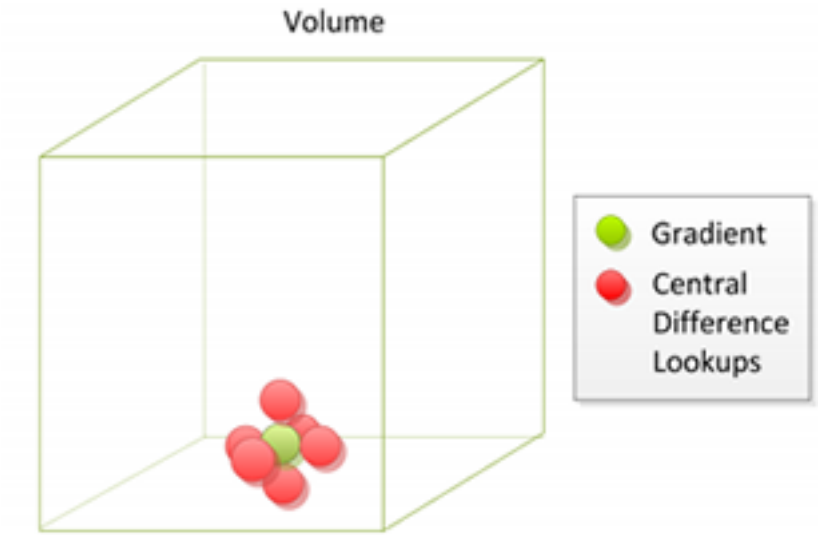

Figure 3.5: Image representing the sampling of the six neighbouring voxel values [8]

$$\nabla f(\mathbf{x}_i) = \nabla f(x_i, y_j, z_k) \approx$$

$$\left[\frac{1}{2}\left[f(x_{i+1}, y_j, z_k) - f(x_{i-1}, y_j, z_k)\right],\right.$$

$$\frac{1}{2}\left[f(x_i, y_{j+1}, z_k) - f(x_i, y_{j-1}, z_k)\right],$$

$$\left.\frac{1}{2}\left[f(x_i, y_j, z_{k+1}) - f(x_i, y_j, z_{k-1})\right]\right].$$

Figure 3.6: Normal Vector Equation [9]

Chapter 4

Implementation

This system uses a combination of HTML, CSS, Javacsript and WebGL. Two helper files for WebGL have been included in the project. The first file is a WebGL utilities Javascript file made by Google and available from Khonos [35]. This file includes functions to initialise the WebGL and to check if a browser's WebGL compatible. The second file is also a Javascript file available from the Google code site [36] which efficiently handles matrix and vector operations.

4.1 Web Worker

As mentioned in section 3.3, a web worker is spawned to load the volume data into the application progressively without freezing the application. In this section, the details of how the data is loaded, parsed and used to generate the textures shall be discussed.

4.1.1 Web Worker Initialisation

To initialise the web worker thread the application calls the Javascript worker constructor and passes in the name of the Javascript worker file, in this case DataLoader.js. This web worker file must be located in the same folder as the HTML file. It is important to note that the worker thread does not have any access to the DOM. The DOM or Document Object Model is the convention for interacting with the different elements on a web page. Since the worker has no access to the DOM, it must communicate with the main thread through a messaging system.

After creating the worker in Javascript, a function is set to the "onmessage" attribute of the worker. This means that a function will be called in the event the worker passes a message to the main thread. Finally, a message is sent to the worker to inform it to start its tasks. This start up message contains a JSON object which contains the file path to the files which the worker has to load and the quantity of these files. JSON which stands for JavaScript Object Notation is a text based standard for exchanging data.

```
1
2  var worker = new Worker("DataLoader.js");
3
4  worker.onmessage = function (event) {
5
6          switch (event.data.type) {
7           case "print":
8                  console.log(event.data.message);          //For
                      printing debug information since the worker
                      has no access to the console
9                  break;
10          case "makeTexture": //Worker has sent values to
               generate a texture - pass these values onto
               appropiate function
11                 CreateTextureFromWorkerArray(....);
12                 break;
13         }
14 };
15
16  worker.postMessage({ filePath: "head/CThead.", fileAmount:
       numberOfFiles }); // starts the worker
```

4.1.2 Loading the data

In the web worker file there is an on message function for when the main thread sends the worker a message. When the worker receives a message it triggers the file loading

system. The worker uses the information passed in the message to load in the correct data files. To give the user a more visually complete dataset while loading, the files are accessed using a divide and conquer approach. The first files downloaded are the ones whose index is divisible by half the amount of files, then a quarter, then a 8 and all the way down to files that are divisible by one. When the data contained in the file is returned, it is sent to a function to handle the data which then sends it on to be parsed.

```
1
2  onmessage = function (event) {
3
4  for (var i = 0; i < fileAmount; i++) {
5          if (i % Math.round(fileAmount/2) == 0) {
6              client = new XMLHttpRequest();
7              client.onreadystatechange = handler;
8              client.open("GET", "volumedata/" + pathName + (i +
                   1), false);
9              client.overrideMimeType('text/plain; charset=UTF-16
                   BE BOM'); //Load is as 16 bit Big Endian
10             client.send();
11         }
12     }
13     for (var i = 0; i < fileAmount; i++) {
14         if (i %  Math.round(fileAmount/4) == 0 && i %  Math.
               round(fileAmount/2) != 0) {
15                 //XMLHttpRequest
16         }}
17 ....
18  for (var i = 0; i < fileAmount; i++) { //loads in all files
       that have been skipped by the dividing algorithm
19         if (i %  Math.round(fileAmount/fileAmount) == 0 && i %
               Math.round(fileAmount/2) != 0 && ...) {
20                 //XMLHttpRequest
21 }  }
22 return; }
```

4.1.3 Parsing the Data and Creating the First Stacks Texture Files

After the message handler is called and no error was reported then the file data is passed on into another function. One of the functions main tasks is to parse the data and insert it into a 3D array which contains information about all the voxels in the data set. The other major responsibility of this task is to pass arrays containing texture information about the main axes textures to the application.

4.1.4 Creating the Perpendicular Stacks Textures

When all the volume files have been downloaded and the 3D array is complete, the alternate axes textures are then computed. This takes place in the worker thread as well. Both axes textures are created at the same time using three nested for loops. Two arrays of pixel data are computed. Each array has a length which is equal to the number of files by the relative resolution of the data volume data. For example with the skull data that is 113 slices of 256 x 256, the output texture array for the alternative axes are 113 by 256.

```
1  for (var k = 0; k < 256; k++) {
2          for (var i = 0; i < fileAmount; i++) {
3                  for (var l = 0; l < 256; l++) {
4                          pixels1[i][l] = volumedata[i][k][l];
5                          pixels2[i][l] = volumedata[i][l][k];
6                  }
7          }
8          postMessage({ type: "makeTexture", direction: "
              Xdirection", pixelData: pixels1, textureIndex: k });
9          postMessage({ type: "makeTexture", direction: "
              Ydirection", pixelData: pixels2, textureIndex: k });
10 }
```

4.2 Rendering the Volume

As mentioned earlier in the design section, the application renders the volume using the 2D texture stacking approach. When the user first loads up the web page, the HTML body element triggers a Javascript function which starts creating the application. After setting up some basic settings for the web application, the Javascript code will call the function to create the planes for each axes.

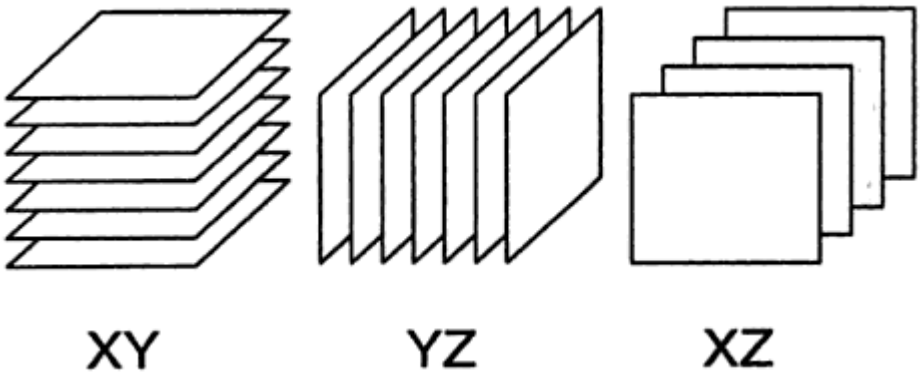

Figure 4.1: Graphic of 2D Texture Stacks from [10]

Each plane has various attributes associated with it including the central position and the texture that should be mapped to it. The primitives are sorted and drawn in back to front order. To draw the volume correctly the depth buffer must be disabled and the WebGL blend function enabled. This allows for the overlapping of different planes without obscuring the planes that are further away from the viewing plane and the correct blending of the rendered alpha values. The blend function parameters are set to the source alpha and one minus the source alpha.

A variable is used to keep track of the current state and which of the stacks to draw. This variable is changed when the orientation of the view plane becomes greater than 45 degrees off the orientation of the stacks being rendered.

Originally when only one stack had been created during the implementation phase, back to front rendering of the polygonal primitives was accomplished used the quick sort algorithm. It became apparent however that when all the stacks are created the quick sort algorithm could be replaced by a computational less expensive method. The technique involves reversing the stack array if it's being rendered from the opposite

direction of where it was previously displayed from. For example, if a stack was being viewed from the top down and the camera position moved to be viewing the stack from the bottom up then the stack order would be reversed, consequently reversing the order it's drawn in.

4.3 Enhancements

In this section, the application enhancements discussed earlier in chapter 3.5 shall be described. A detailed description of their implementation shall be provided alongside example code snippets from the application.

4.3.1 Volume Clipping

To implement the volume clipping described in section 3.5.1. a combination of CPU and GPU methods are used. First, when drawing the texture stack, the application looks at the slice plane that is parallel to the stack being drawn. If the position of the cutting plane is greater than the position of the slice to be drawn then that slice can be drawn. If the slice lies at a position greater than the cutting plane, then the slice isn't drawn.

For clipping planes that are perpendicular to the slices being rendered then two uniform variables are passed to the GLSL shaders.

```
1 gl.uniform1f(shaderProgram.cutPlane1Uniform, perpSlice1);
2 gl.uniform1f(shaderProgram.cutPlane2Uniform, perpSlice2);
```

The shader receives these variables and uses their values to decide which volume fragments should be drawn. Since the variables passed in are resized based on the clipping planes position relative to the volume, to equal a value between zero and one, they corespond directly to the UV texture mapping coordinate of the volume planes to be drawn. The shader then sets the opacity value of every fragment that is positioned before the clipping planes position to be zero, consequently rendering them invisible.

```
if(TextureCoord.s < perpSlice1 || vTextureCoord.t < perpSlice2)
{
        Color.a = 0.0;
}
```

4.3.2 Transfer Function

The transfer function is created by inputting three separate texture files, one for each example of a transfer function. The user may choose which one to use through the user interface on screen. The transfer function texture selected is then bound to a texture channel (in this case channel one). A uniform integer representing the texture channel from which to access the transfer function texture is then sent to the GLSL shader.

```
gl.activeTexture(gl.TEXTURE1);
gl.bindTexture(gl.TEXTURE_2D, transferfunctionTexture);
gl.uniform1i(shaderProgram.transferUniform, 1);
```

The GLSL fragment shader retrieves the appropriate colour from the transfer function texture using a 2D texture lookup. The alpha value of the texture to be drawn corresponds to the density of the volume at that pixel. It is this alpha value that is used as the position variable for the 2D texture lookup.

```
vec4 transferColor = texture2D(transferSampler, vec2(alpha,
   0.0));
```

4.3.3 Phong Lighting though Iso Surface Extraction

This section describes the implementation of the Phong lighting method detailed in section 3.5.3. It calculates each fragments normal vector by examining the values of the six neighbouring voxels.

There are two possible ways of implementing the normal calculations. Technique one involves processing all the gradients at once and storing them in the red, green

and blue channels of the texture. This methods advantage is that the gradients only need to be estimated once but as a result the method is inflexible and not suited for animated voxels. Technique two is implemented by calculating gradients on the fly utilising the GPU. The advantage to this method is that it takes up less CPU time and is animation friendly.

Calculating gradients on the fly was the variation chosen to be implemented in this application. The GLSL shader calculates the x and y gradients by sampling the alpha at various positions in the current slice texture.

```
1 g1.x = texture2D(uSampler, vec2(vTextureCoord.s - sampleSize,
     vTextureCoord.t)).a;
2 g2.x = texture2D(uSampler, vec2(vTextureCoord.s + sampleSize,
     vTextureCoord.t)).a;
3 g1.y = texture2D(uSampler, vec2(vTextureCoord.s, vTextureCoord.
     t - sampleSize)).a;
4 g2.y = texture2D(uSampler, vec2(vTextureCoord.s, vTextureCoord.
     t + sampleSize)).a;
```

The z value in the normal vector is calculated using additional textures. Where the x and y normals lie on the texture to be drawn; the z values needed are located in the textures of the planes that are scheduled to be drawn before and after the current plane. This is accomplished by passing in the before and after textures to be drawn by binding them to the second and third texture channels. Both integer uniforms referring to the texture channel to sample from are then passed into the GLSL shader.

```
1 gl.activeTexture(gl.TEXTURE2);
2 gl.bindTexture(gl.TEXTURE_2D, slices[current - 1].texture);
3 gl.activeTexture(gl.TEXTURE3);
4 gl.bindTexture(gl.TEXTURE_2D, slices[current + 1].texture);
5 gl.uniform1i(shaderProgram.previousTextureUniform, 2);
6 gl.uniform1i(shaderProgram.nextTextureUniform, 3);
```

The GLSL file then calculates the z values in a similar way to the x and y values; the main difference being that the values are sampled from different textures. The current

fragments normal gradient is then created by normalising the the vector equaling the next gradient vector minus the previous one.

```
1 g1.z = texture2D(nextTexture, vec2(vTextureCoord.s,
     vTextureCoord.t)).a;
2 g2.z = texture2D(previousTexture, vec2(vTextureCoord.s,
     vTextureCoord.t)).a;
3 normal = normalize(g2 - g1);
```

4.3.4 UI and User Controls

The UI is implemented using a combination of HTML5 and CSS. The general layout of the site is generated using the 960 grid system [37]. The 960 grid system is a collection of CSS files that are free to use which help web developers rapidly prototype site layouts.

The menu on the page is created using HTML5 input devices such as sliders and check boxes. Each input element has a Javascript method attached to it. Should the value in the input change then the Javascript function is called to update the relative parameters to the new value.

Chapter 5

Evaluation

This chapter will present testing results from running the application described in the paper. The object of this dissertation was to create an in browser web system to display 3D volume data to the average internet user without the use of any additional plugins. This system allows the user to clearly view the volume while presenting the user with various options to manipulate the display of the volume to enhance the users understanding of the dataset.

As one of the aims of this dataset is to enable the average user to view the volume all the results are tested on two computers. One is a average household laptop Dell Inspiron 1520, with the following specifications:

- Windows 7 E 32 bit.
- Nvidia GeForce 8600 GT Graphics Card, 256MB of VRAM.
- Intel Core Duo CPU, T7500 2.2 Ghz processor.
- 3.5GB RAM.

The other computer that the application will be tested on is a top of the range modern desktop PC the Dell Precision T3500, with the following specifications:

- Windows 7 Enterprise 32 bit.
- Nvidia Quatro FX 580, 512MB of VRAM.

Table 5.1: Google Chrome Performance Table (Desktop)

Iteration Number	Download Time	Frame Rate	Texture Generation
1	12.682	38 fps	21.998
2	8.454	39 fps	17.908
3	8.173	34 fps	17.526
4	8.218	29 fps	17.383
5	8.902	34 fps	18.367
Average	9.286	43.8 fps	18.636

- Intel x86 Core Duo CPU, T7500 2.6 Ghz processor.
- 4GB RAM.

The WebGL enabled browser installed on both computers for testing was Google's Chrome Version 13.x and Mozilla's Firefox 6.0. Both browsers are the newest stable releases of the software. The application was not tested using the Safari browser as a result of the fact that Safaris WebGL is only supported on Mac computers with the Snow Leopard Operating System (OS X 10.6).

5.0.5 Performance Results

The following results are taken from test runs of five iterations from the Chrome and Firefox browser tested on the modern desktop. The internet connection used had a download speed of 93mbps (mega bytes per second) and a upload speed of 25.84. Table 5.1 details the results of the test for the Chrome browser, while table 5.2 presents the results of running the application through the Firefox browser. The cache of each browser was emptied before running each test to ensure there was no error in calculating the download times. The last column represents the averaged results.

Chrome overall gives a excellent performance, but it is interesting to note there is a obvious difference in efficiency between the two browsers. Currently, Google's Chrome has a reputation for being the fastest browser and the results confirm this. Where Chrome maintains an average fps of 34.8, Firefox only executes at a average of 15.6 fps. That means that Chrome performs on average at more than twice the frame rate of Firefox. This is clearly noticeably during testing.

Table 5.2: Mozilla Firefox Performance Table (Desktop)

Iteration Number	Download Time	Frame Rate	Texture Generation
1	26.896	19 fps	31.748
2	26.032	13 fps	31.056
3	27.024	17 fps	32.017
4	26.236	13 fps	31.095
5	34.081	16 fps	38.893
Average	28.0718	15.6 fps	32.294

This is possibly due to a difference between how the two browsers work. In Firefox, there is a single process for each window; if a window has multiple tabs open then they all become part of the one process. In Firefox, if one tab freezes due to a lengthy task, all other tabs stall with it. Google's Chrome handles its web pages quite differently, opening multiple processes for each window depending on the amount of tabs currently open.

Similar rates are also seen in the other categories. Where Chrome takes a average of 9.2858 seconds to download and parse 113 files (14.1MB) over a 93mbps internet connection, Firefox takes 28.0718 seconds over the same connection. This is a increase of almost 20 seconds, a substantial time difference when it comes to browsing the internet.

The graphs also show a large difference in the time it takes for each browser to generate all the textures, 18.6364 for Chrome and 32.9638 for Firefox. Although this is inclusive of the amount of time taken to download the files so Firefox consequently takes a much longer time then Chrome automatically.

Figure 5.1 illustrates the main differences between Firefox and Chrome. It shows Chrome to consistently be the better browser for viewing the application, although these results do not match the current state of the web. As shown in figure 1.5, Chrome (as off June 2011) only has 16.5% of the browser usage whereas Firefox is the more popular browser at 25.4%.

The next set of performance tests ran was using the Dell laptop to test the system on a average modern day laptop. This is considered an important test since the part of the motivation of this dissertation was to enable the average user to utilise the application. These tests were carried out over a DSL internet connection with download speeds of 11.45 mpbs and upload speeds of 0.62 mbps.

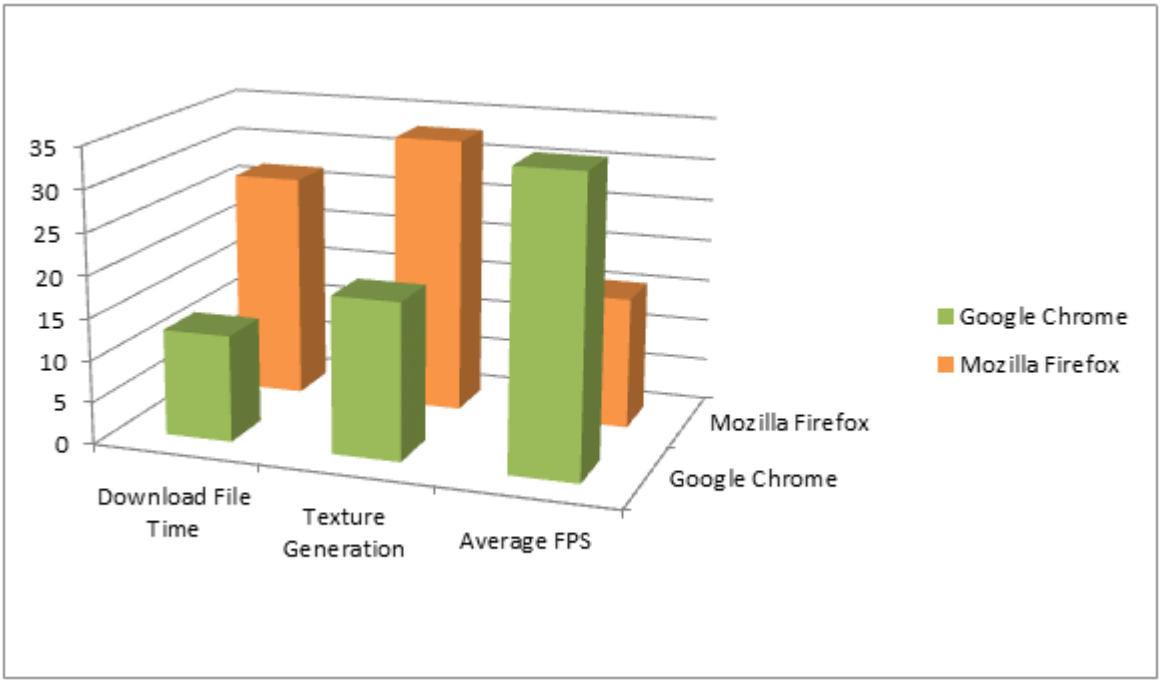

Figure 5.1: Chrome and Firefox Comparison Graph using Desktop

Tables 5.3 and 5.4 detail the results of testing the application on the Dell laptop with both browsers. Evident in these graphs are similar patterns to the tests carried out on a top of the range desktop with a high speed internet connection. The results again demonstrate that the Google Chrome browser gives a much better performance when running the application than the Firefox browser. Although the download speeds are much slower on the laptop, this was expected since the laptop was tested on a slower internet connection. The user interface remained interactive throughout all tests and there were no unresponsive browser warnings issued. These test results show

Table 5.3: Laptop Performance Results for Google Chrome

Iteration Number	Download Time	Frame Rate	Texture Generation
1	31.951	44 fps	54.259
2	32.266	49 fps	53.499
3	37.775	40 fps	60.003
4	30.927	44 fps	50.647
5	29.9	44 fps	50.046
Average	32.564	44.2 fps	53.6908

Table 5.4: Laptop Performance Results for Mozilla Firefox

Iteration Number	Download Time	Frame Rate	Texture Generation
1	81.975	12 fps	88.515
2	74.597	13 fps	81.159
3	54.635	9 fps	61.563
4	53.262	11 fps	60.182
5	60.35	11 fps	67.229
Average	64.964	11.2 fps	71.73

that the application is suitable for real time use with both browsers, although better performance is achieved with Google Chrome.

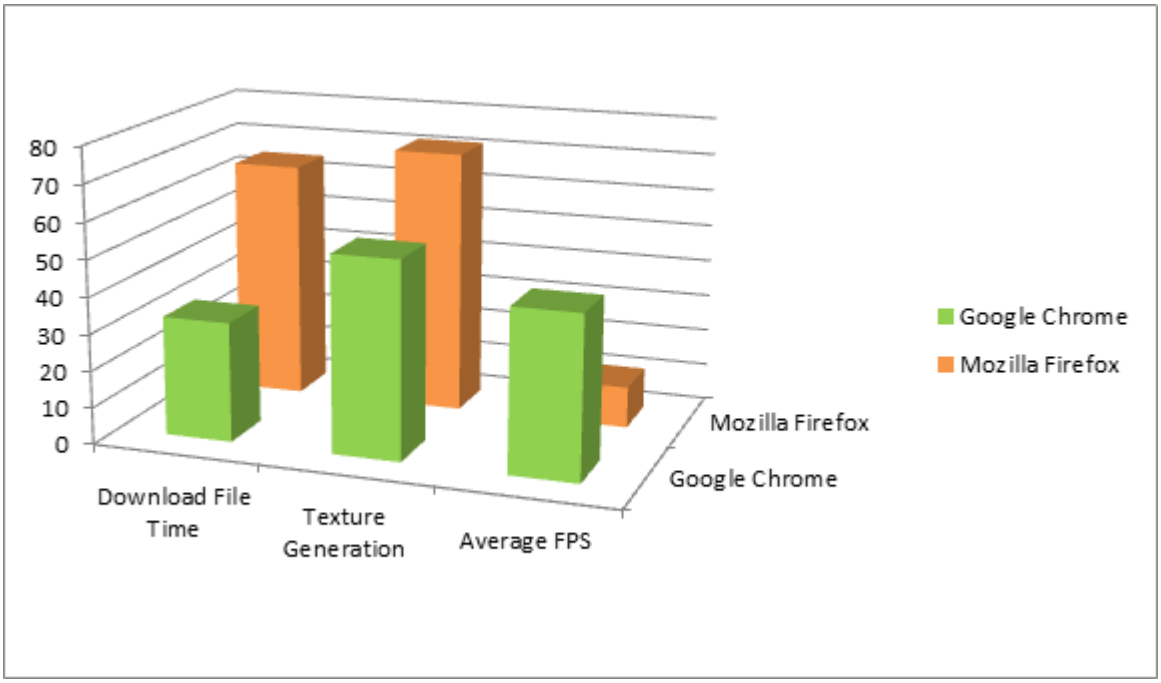

Figure 5.2: Chrome and Firefox Comparison Graph using laptop

5.0.6 Visual Results

In this section the visual appearance of the volume will be evaluated.This is an important aspect of the application as rendering the volume without artefacts and in a manner which allows the data to be understood correctly are paramount considerations

in medical and scientific fields.

The overall appearance of the volume is satisfactory to the aim of the application. It clearly illustrates the volume being displayed and gives the user multiple options to enhance the visuals depending on their requirements. However, there are various features that could possibly be improved upon.

The gradient, when turned on, can be noisy. This is due to how it's rendered on the fly i.e. rendered during run time. If the gradients were generated at load time into the red, green and blue colour channels of the volume this would allow for a smoothing algorithm to be used on the colour channels of the texture to filter out the noise.

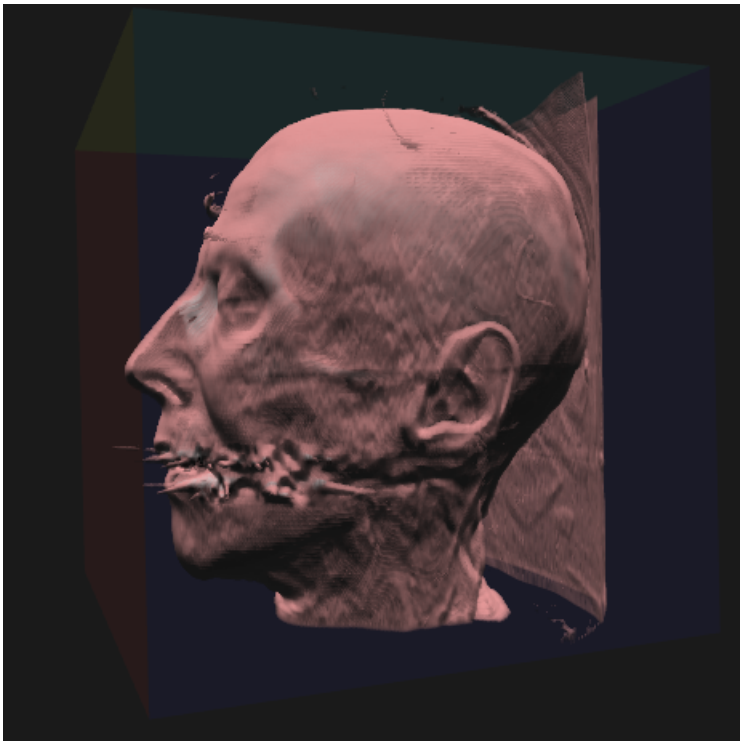

Figure 5.3: Noise generated from on the fly gradient calculations

Another concern is the flicker of textures that can be observed when switching from one texture stack to another. This is a rendering problem inherent in volume rendering through 2D texture stacking. Very slight visual shifts can be observed when the angle passes the threshold limit for the current texture stack and the new, more appropriate stack is drawn.

5.0.7 User Interface

The user interface is intuitive and allows the user to easily edit rendering variables. It presents the user with several parameters that can be modified to allow the user to see the volume, or certain sections of the volume, as they require.

Customisable parameters provided include three slider input bars to edit the position of the clipping planes, two slider bars to manipulate opacity settings, a checkbox item for activating or deactivating the gradient and a graphical button system for choosing the current pre-set transfer function. To rotate the volume the client can use either the mouse or the W, A, S and D keys on the keyboard as input.

Chapter 6

Conclusions

In this dissertation, an application for progressive volume rendering was proposed. The main aim was to present a system that could work in modern web browsers. This is achieved by utilising HTML5 and WebGL in WebGL enabled browsers. The volume data to be rendered is downloaded to the local desktop from the server, where the data is then transformed into textures representing the volume and rendered on screen.

The system proposed in the dissertation accomplish these aims in full. The main challenge presented in the application was the implementation of the volume renderer using data files that needed to be progressively streamed in from a server online. Not only is the system capable of loading in the data, but it also does so maintaining a real time frame rate while presenting the user with a UI they which allows them to modify the display of the volume instantaneously.

Presently, WebGL is still in the development stage of its first version and itis important to note that it is not yet supported on all the major browsers. However, when WebGL is supported as standard in browsers, a volume rendering application such as the one presented in this dissertation could be a valuable tool in medicine and science alike, for examining and dissecting various 3D volumes.

6.1 Future Work

Volume Data Compression

The application could be improved with the use of a method of compression for the volume data. Methods like those mentioned in [38] and in [39] help to shrink the size of the storage needed for the volume infomration, making it more efficient and consequently speeding up the transmission of the volume data over the internet.

Sampling Rates / Level of Detail

The application could be further improved with the possibility of level-of-detail volume rendering. Methods similar to those listed in [40] could be examined to allow the use of adaptive sampling of the data. Potentially providing a method to load in larger datasets that otherwise would be too CPU intensive to store.

Different Rendering Techniques

Research into different volume rendering methods adaptable for WebGL could be extended. The possibiility of using the splatting method in [22] or the Shear-Warp method in [3] should be investigated.

Region Selecting

Another area of research that could be examined is the generation of region selection when using 2D texture stacks. Although there has been some research into region selection in 3D textures [41] and polygonal representations of the data [41] not much research has gone into the implementation of these methods with 2D texture stacking.

Rendering Techniques

Futher research to benefit the application could include the examining of rendering techniques such as edge detection [42] and curvature shading [43] to enhance the visual representation of the data.

Appendix A

Application Screen Shots

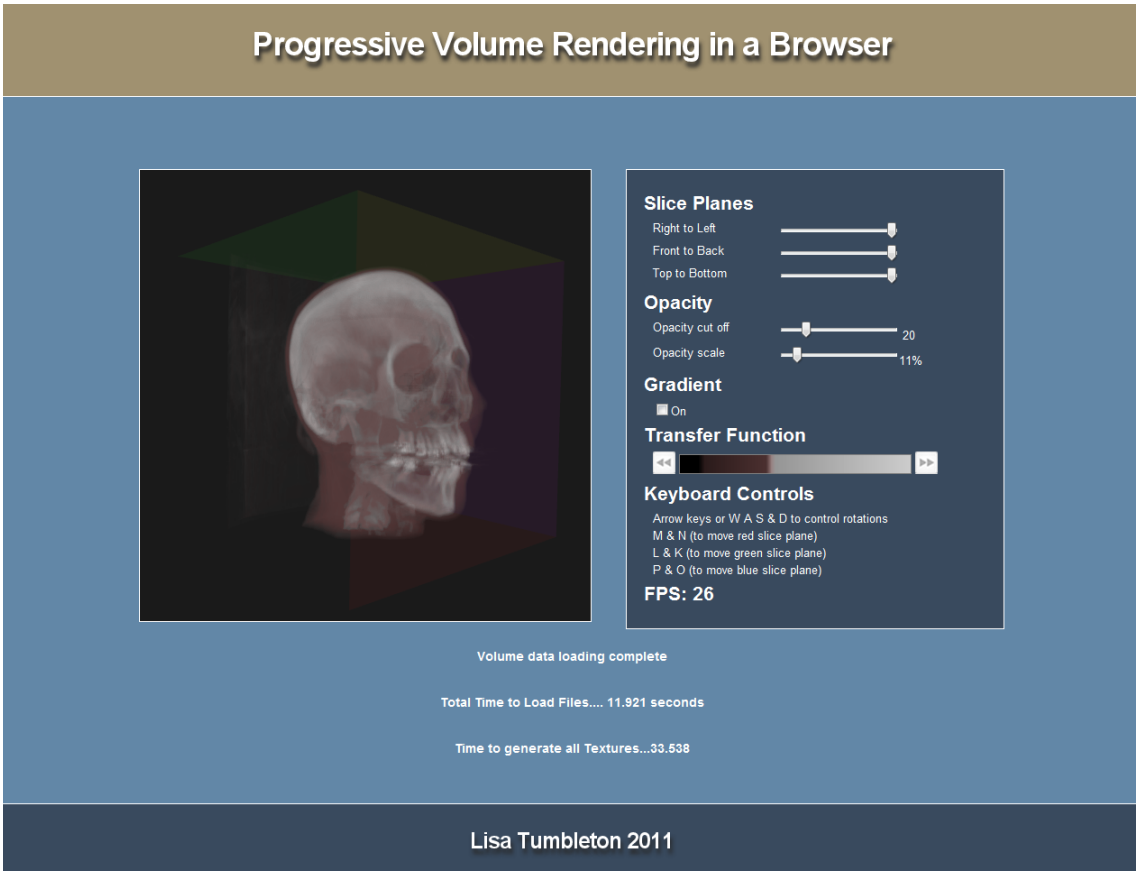

Figure A.1: Application Screen Shot

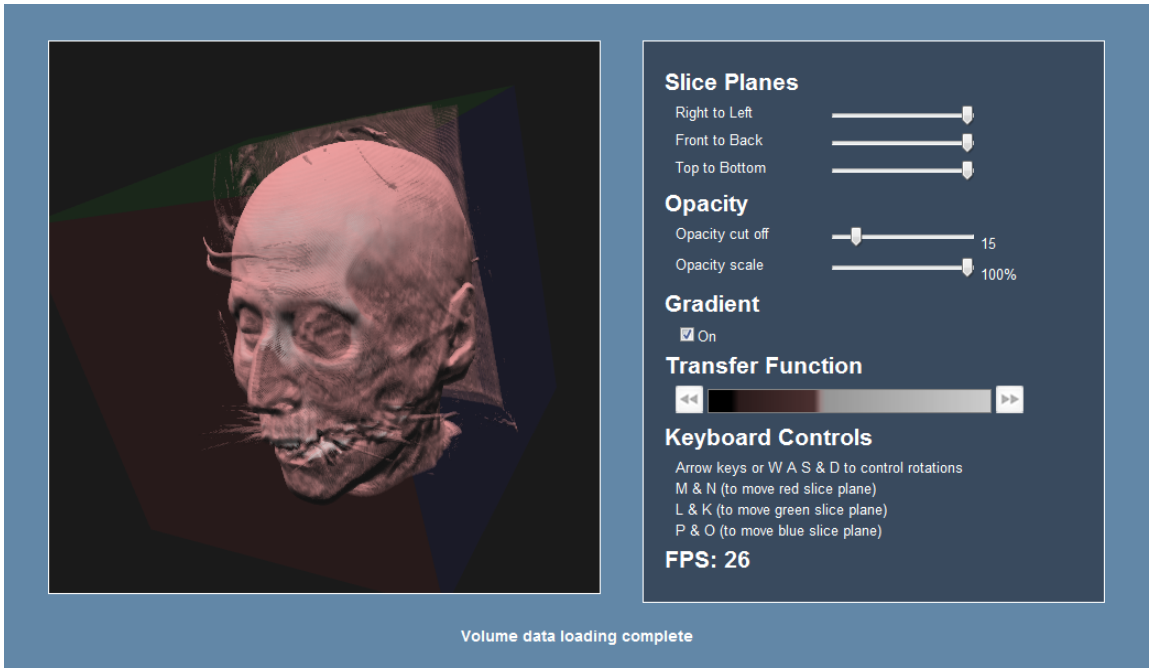

Figure A.2: Application Screen Shot, with Phong shading

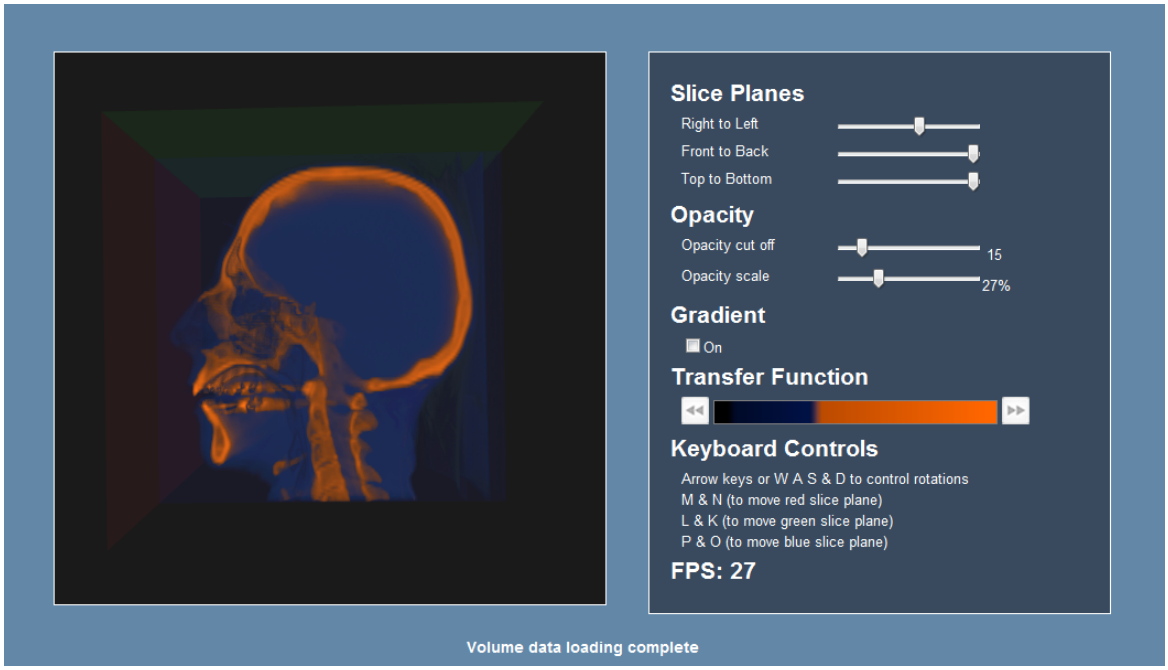

Figure A.3: Application Screen Shot, with clipping plane

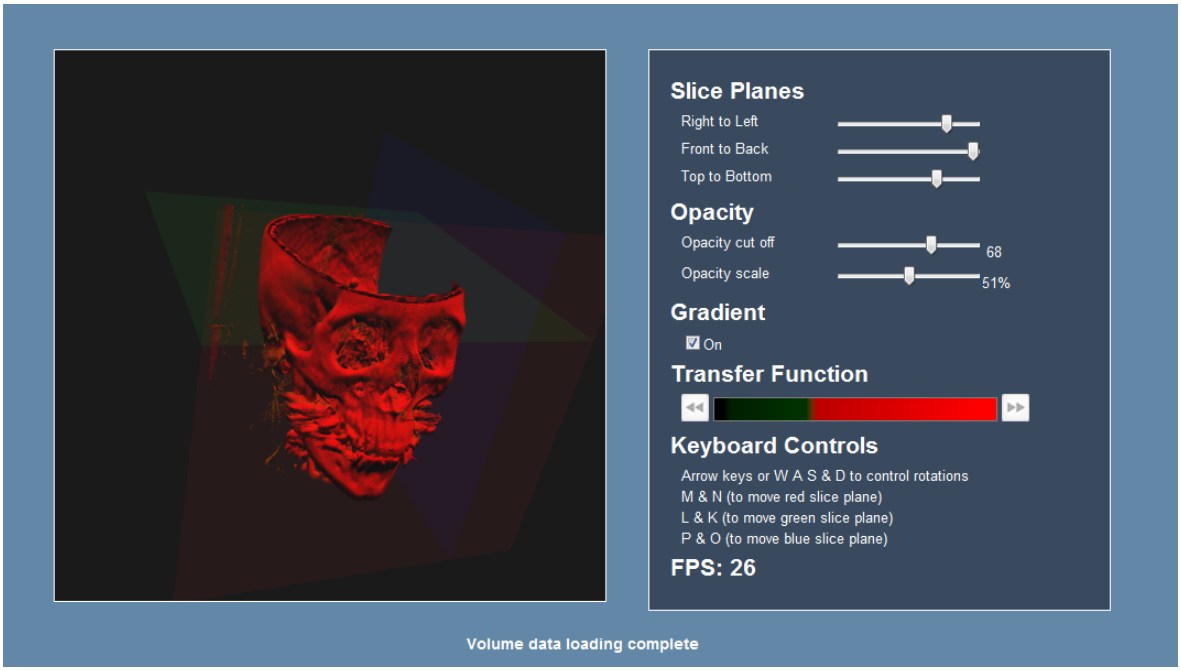

Figure A.4: Application Screen Shot, with Phone shading and multiple clipping planes

Bibliography

[1] H. Fuchs, Z. Kedem, and S. Uselton, "Optimal surface reconstruction from planar contours," *Communications of the ACM*, vol. 20, no. 10, pp. 693–702, 1977.

[2] D. Laur and P. Hanrahan, "Hierarchical splatting: A progressive refinement algorithm for volume rendering," in *ACM SIGGRAPH Computer Graphics*, vol. 25, pp. 285–288, ACM, 1991.

[3] P. Lacroute and M. Levoy, "Fast volume rendering using a shear-warp factorization of the viewing transformation," in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 451–458, ACM, 1994.

[4] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *Proceedings of the 1994 symposium on Volume visualization*, pp. 91–98, ACM, 1994.

[5] R. Westermann and T. Ertl, "Efficiently using graphics hardware in volume rendering applications," in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 169–177, ACM, 1998.

[6] M. Levoy, "Efficient ray tracing of volume data," *ACM Transactions on Graphics (TOG)*, vol. 9, no. 3, pp. 245–261, 1990.

[7] K. Engel and T. Ertl, "Texture-based volume visualization for multiple users on the world wide web," in *5th Eurographics Workshop on Virtual Environments*, pp. 115–124, Citeseer, 1999.

[8] C. J. da Cruz Ramalhao, "Painterly stylization of real-time volume rendering," 2010.

[9] M. Levoy, "Display of surfaces from volume data," *Computer Graphics and Applications, IEEE*, vol. 8, no. 3, pp. 29–37, 1988.

[10] O. Hendin, N. John, and O. Shochet, "Medical volume rendering over the www using vrml and java.," *Studies in health technology and informatics*, vol. 50, p. 34, 1998.

[11] R. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," in *ACM Siggraph Computer Graphics*, vol. 22, pp. 65–74, ACM, 1988.

[12] T. Berners-Lee, "The worldwideweb browser," tech. rep., W3 Schools, 2011. http://www.w3.org/People/Berners-Lee/WorldWideWeb.

[13] "World internet users and population stats," tech. rep., Miniwatts Marketing Group, 2011. http://www.internetworldstats.com/stats.htm.

[14] M. Harris, W. Baxter, T. Scheuermann, and A. Lastra, "Simulation of cloud dynamics on graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 92–101, Eurographics Association, 2003.

[15] M. Stytz, G. Frieder, and O. Frieder, "Three-dimensional medical imaging: algorithms and computer systems," *ACM Computing Surveys (CSUR)*, vol. 23, no. 4, pp. 421–499, 1991.

[16] S. Horiil, F. Prior, W. Bidgood, C. Parisot, and G. Claeys, "Dicom: an introduction to the standard," *Disponível na WWW, URL: http://www. dicomanalyser. co. uk/html/introduction. htm. Último acesso em*, vol. 20, 2002.

[17] E. Keppel, "Approximating complex surfaces by triangulation of contour lines," *IBM Journal of Research and Development*, vol. 19, no. 1, pp. 2–11, 1975.

[18] G. Herman and J. Udupa, "Display of 3-d digital images: Computational foundations and medical applications," *Computer Graphics and Applications, IEEE*, vol. 3, no. 5, pp. 39–46, 1983.

[19] W. Lorensen and H. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *ACM Siggraph Computer Graphics*, vol. 21, no. 4, pp. 163–169, 1987.

[20] M. Levoy and T. Whitted, "The use of points as a display primitive," *Tech. Report85-022, University of North Carolina at Chapel Hill*, 1985.

[21] L. Kobbelt, M. Botsch, U. Schwanecke, and H. Seidel, "Feature sensitive surface extraction from volume data," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 57–66, ACM, 2001.

[22] L. Westover, "Footprint evaluation for volume rendering," *ACM Siggraph Computer Graphics*, vol. 24, no. 4, pp. 367–376, 1990.

[23] J. Kruger and R. Westermann, "Acceleration techniques for gpu-based volume rendering," 2003.

[24] T. Whitted, "An improved illumination model for shaded display," *Communications of the ACM*, vol. 23, no. 6, pp. 343–349, 1980.

[25] S. Laine and T. Karras, "Efficient sparse voxel octrees," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pp. 55–63, ACM, 2010.

[26] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering," in *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pp. 15–22, ACM, 2009.

[27] S. Ortiz, "Is 3d finally ready for the web?," *Computer*, vol. 43, no. 1, pp. 14–16, 2010.

[28] "Html5 specification, canvas section," tech. rep., W3 Schools, 2011. http://dev.w3.org/html5/spec/Overview.html.

[29] "Web gl specification," tech. rep., Khronos Group, 2011. https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/doc/spec/WebGL-spec.html.

[30] A. Borsos, "Webgl as an alternative to platform-specific 3d apis," 2010.

[31] "Script run time," tech. rep., MozillaZine, 2011. http://kb.mozillazine.org.

[32] "Stanford volume data repository," tech. rep., Stanford Computer Graphics Laboratory, 2011. http://graphics.stanford.edu/data/voldata/.

[33] E. K. F. DREBIN, ROBERT A. and D. MAGID, "Volumetric three-dimensional image rendering: Thresholding vs. non-thresholding techniques," *Radiology*, vol. 165, no. 1, p. 131, 1987.

[34] B. Phong, "Illumination for computer-generated images.," tech. rep., DTIC Document, 1973.

[35] G. Inc., "Webgl utilities helper," tech. rep., Khronos Organisation, 2011. https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/sdk/demos/common/webgl-utils.js.

[36] Tojiro, "glmatrix-0.9.5.min.js," tech. rep., Google, 2011. http://code.google.com/p/glmatrix/.

[37] N. Smith, "960 grid system," tech. rep., 2011. http://960.gs/.

[38] J. Fowler and R. Yagel, "Lossless compression of volume data," in *Proceedings of the 1994 symposium on Volume visualization*, pp. 43–50, ACM, 1994.

[39] K. Nguyen and D. Saupe, "Rapid high quality compression of volume data for visualization," in *Computer Graphics Forum*, vol. 20, pp. 49–57, Wiley Online Library, 2001.

[40] M. Weiler, R. Westermann, C. Hansen, K. Zimmermann, and T. Ertl, "Level-of-detail volume rendering via 3d textures," in *Proceedings of the 2000 IEEE symposium on Volume visualization*, pp. 7–13, ACM, 2000.

[41] T. Yoo, U. Neumann, H. Fuchs, S. Pizer, T. Cullip, J. Rhoades, and R. Whitaker, "Achieving direct volume visualization with interactive semantic region selection," in *Proceedings of the 2nd conference on Visualization'91*, pp. 58–65, IEEE Computer Society Press, 1991.

[42] J. Canny, "A computational approach to edge detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, no. 6, pp. 679–698, 1986.

[43] M. Hadwiger, C. Sigg, H. Scharsach, K. B
"uhler, and M. Gross, "Real-time ray-casting and advanced shading of discrete isosurfaces," in *Computer Graphics Forum*, vol. 24, pp. 303–312, Wiley Online Library, 2005.